Faculty of sciences

UMONS
University of Mons

Reproducibility in Software Engineering

Master's Thesis in Computer Science

Author : Pol Dellaiera

Supervisor : Prof. Dr. Tom Mens

Academic year : 2023 - 2024

Submission date : 12 June 2024

Build date : Fri Jul 5 15:45:28 UTC 2024

DOI : 10.5281/zenodo.12666899

Revision : 90c453e

Faculty of Sciences ◆ University of Mons ◆ 20, Place du Parc ◆ B-7000 Mons

This page is intentionally left blank.

I confirm that this Master's thesis is my own work and I have documented all sources and material used.

Mons, 12 June 2024

Pol Dellaiera

This page is intentionally left blank.

Abstract

The concept of reproducibility has long been a cornerstone in scientific research, ensuring that results are robust, repeatable, and can be independently verified. This concept has been extended to computer science, focusing on the ability to recreate identical software artefacts. However, the importance of reproducibility in software engineering is often overlooked, leading to challenges in the validation, security, and reliability of software products.

This master's thesis aims to investigate the current state of reproducibility in software engineering, exploring both the barriers and potential solutions to making software more reproducible and raising awareness. It identifies key factors that impede reproducibility such as inconsistent environments, lack of standardisation, and incomplete documentation. To tackle these issues, I propose an empirical comparison of tools facilitating software reproducibility.

To provide a comprehensive assessment of reproducibility in software engineering, this study adopts a methodology that involves a hands-on evaluation of four different methods and tools. Through a systematic evaluation of these tools, this research seeks to determine their effectiveness in establishing and maintaining identical software environments and builds.

This study contributes to academic knowledge and offers practical insights that could influence future software development protocols and standards.

This page is intentionally left blank.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Professor Tom Mens. I am incredibly thankful for his availability and guidance throughout my studies. It was a great honour to receive his proposal to supervise my research in my final year. Beyond that, he generously allowed me to suggest topics that piqued my interest, ultimately enabling me to focus on a subject that I am truly passionate about.

I must also express my heartfelt appreciation for my girlfriend, Sandra. Her endless patience and emotional support have been a constant source of strength, motivation and inspiration for me. However, it is with a touch of melancholy that I acknowledge the sacrifices we've made in our personal lives in order to pursue this overdue academic endeavour that I should have completed twenty years ago.

I would like to thank my family and *in-real-life* friends for their continuous support and encouragement throughout these last years. Your belief in me has provided the foundation upon which this work stands. I am also deeply grateful to my *online* friends, especially within the Typst and Nix communities, for their tremendous support and constant source of solutions, inspiration and motivation.

A special mention is deserved for Izumi, my cat, who was a constant companion and source of comfort over the last decade. His loss was a profound sorrow, and I deeply miss his presence. The memories of the countless hours he spent by my side, offering silent support during my work, have left an indelible mark.

Finally, I would like to express my sincere thanks to all the participants in this research. I am particularly grateful to my colleagues at European Commission, who courageously and continuously supported me while remaining unaware of my academic activities. Your valuable feedback has greatly contributed to the development of some parts of this master's thesis. In fact, your lack of awareness helped me understand the barriers to implementing software reproducibility from the very beginning and in a real professional context. Each piece of feedback has been instrumental in helping me better understand and improve communication about this concept.

Thank you so much!

Pol Dellaiera.

This page is intentionally left blank.

Accessibility

This master thesis has been written with a focus on accessibility to ensure it can be easily read and understood by a diverse audience, including individuals with disabilities. The following measures have been taken to enhance accessibility:

- Links: all hyperlinks within this document are underlined and clearly distinguishable from regular text. This visual cue helps users identify clickable links easily.
- Symbols and notation: specific symbols and notation have been used consistently throughout the document to aid comprehension. Mathematical symbols, special characters, and other notation are presented in a clear and readable manner.
- Text formatting: the document uses high-contrast text formatting and font sizes that are readable across different devices and screen resolutions. The `New Computer Modern` [1] font is used, chosen for its clarity and readability, especially in mathematical and technical contexts.
- Margins: the margins have been alternately adapted to ensure that when the document is printed, it is suitable for binding and easy to read. This consideration enhances the physical accessibility of the printed document.
- Headings and structure: the document is structured with clear headings and subheadings to facilitate navigation. This hierarchical organisation assists readers in quickly finding relevant sections.
- Language and terminology: plain language and concise terminology have been employed to ensure that the content is comprehensible to a broad audience, including those for whom English is not their first language.
- Glossary: a glossary is included, containing the most common abbreviations used throughout the document. This aids readers in quickly understanding the abbreviations and acronyms, improving overall comprehension. Additionally, comprehensive lists of definitions, figures, and tables are provided.
- Accessible file formats: the document is available in multiple file formats, to accommodate various reading preferences and assistive technologies.
- Bibliography links: in line with the Institute of Electrical and Electronics Engineers (IEEE) citation style, numbers in brackets are used to link references in the bibliography. This method provides a clear and consistent way to reference sources, enhancing the readability and accessibility of the document.
- Images: all images in this document are in Scalable Vector Graphics (SVG) format. This vectorial format ensures that images are scalable without loss of quality, providing clear and accessible visuals on different devices and screen resolutions.

Open Source

This master thesis has been developed exclusively using open-source tools. Similar to an open-source project, it was maintained on GitHub but in a private repository [2]. That repository will be made public after the oral defense and necessary internal cleanup. Automated and reproducible builds were managed via GitHub Actions [3], ensuring that a new compiled version was published at each commit to the project. Additionally, I'm planning to publish it on Zenodo [4] too after the necessary formalities are completed.

This work is licenced under a dual license: the Creative Commons Attribution 4.0 International (CC BY 4.0) and the Hippocratic Licence 3.0 (HL3) licences. You are free to share and adapt the material under the terms of the `CC BY 4.0`, provided you give appropriate credit to the original author. You must also use the material in accordance with the ethical guidelines specified in `HL3`, ensuring it is not used to contribute to human rights abuses or other unethical practices. In case of any conflict between the licences, `HL3` will take precedence.

For the purpose of the Chapter 3, an open-source project [5] was created to provide a full transparency on the results shown in that chapter.

Glossary

***CC BY 4.0* – Creative Commons Attribution 4.0 International**: The Creative Commons Attribution 4.0 International License [6] is a widely used license that allows others to distribute, remix, adapt, and build upon your work, even commercially, as long as they credit you for the original creation. This is the most flexible of the CC licenses. 9

***CDN* – Content Delivery Network**: A content delivery network is a system of distributed servers that deliver web content to a user based on the geographic locations of the user, the origin of the webpage and a content delivery server, making the delivery of content more efficient. 74

***CI/CD* – Continuous Integration/Continuous Deployment**: Continuous Integration (CI) is a software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous Deployment (CD) is a software release process that uses automated testing to validate that changes are safe to deploy to production. 10, 52, 66

***CPU* – Central Processing Unit**: The CPU is the primary component of a computer that processes instructions. It runs the operating system and applications, constantly receiving input from the user or active software programs. It processes the data and produces outputs. ARM and X86 are two common CPU architectures. 31, 37, 73

***CRA* – Cyber Resilience Act**: The Cyber Resilience Act [7] is a proposed European Union regulation that aims to improve the cybersecurity of digital products and services. It includes provisions for software supply chain security, incident reporting, and security certification. 37

***CS* – Computer Science**: The discipline of Computer Science includes the study of algorithms and data structures, computer and network design, modelling data and information processes, and artificial intelligence. Computer Science draws some of its foundations from mathematics and engineering and therefore incorporates techniques from areas such as queueing theory, probability and statistics, and electronic circuit design. 16, 18, 19, 24, 25, 30, 31, 33, 34, 35, 46, 81

CycloneDX: CycloneDX [8] is an open-format standard baked by the OWASP foundation and Ecma Technical Committee designed to provide comprehensive and interoperable information about the components used within software projects like software bill of materials and advanced supply chain capabilities for cyber risk reduction. 38

***DSL* – Domain Specific Language**: A domain-specific language is a computer language specialised to a particular application domain. This is in contrast to a general-purpose language, which is broadly applicable across various domains. 70

DevOps: DevOps is a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development life cycle and provide Continuous Integration/ Continuous Deployment (CI/CD). 10, 54

DevSecOps: DevSecOps is an extension of DevOps practices that integrates security (Sec) measures at every stage of the software development lifecycle, ensuring that security is a fundamental aspect of development and operations processes. 38

***FHS* – Filesystem Hierarchy Standard**: The Filesystem Hierarchy Standard is a reference document that describe the conventions used for the layout of Unix-like operating systems. This includes names, locations, and permissions of many file and directories. 58

***HL3* – Hippocratic Licence 3.0**: The Hippocratic Licence 3.0 [9] is a software license that ensures that software is not used to contribute to human rights abuses or other unethical practices. It is designed to protect users and communities from the potential misuse of software. 9

***IEEE* – Institute of Electrical and Electronics Engineers**: The Institute of Electrical and Electronics Engineers [10], established in 1963, is the world's largest technical professional organisation dedicated to advancing technology for the benefit of humanity. It serves as a professional association for electronic engineering, electrical engineering, and related disciplines. 9, 11

***MD5* – Message Digest 5**: The MD5 message-digest algorithm is a widely used hash function producing a 128-bit hash value. MD5 was designed by Ronald Rivest in 1991 to replace an earlier hash function MD4, and was specified in 1992 as RFC 1321. 35, 36

***OCI* – Open Container Initiative**: OCI stands for Open Container Initiative [11], an open governance project for the purpose of creating open industry standards around container formats and runtime. An "OCI image" is a container image that conforms to the OCI image format specification. 54, 61, 62, 69

***OS* – Operating System**: An operating system is system software that manages computer hardware and software resources, and provides common services for computer programs. 26, 31, 37, 56

***PDF* – Portable Document Format**: A file format developed by Adobe in the 1990s to present documents, including text formatting and images, in a manner independent of application software, hardware, and operating systems. 19, 61, 62, 63, 64, 81

***POSIX* – Portable Operating System Interface**: POSIX is a family of standards specified by the IEEE for maintaining compatibility between operating systems. 43, 58

***PURL* – Package URL**: A PURL [12] is a Uniform Resource Locator (URL) string used to identify and locate a software package in a mostly universal and uniform way across programing languages, package managers, packaging conventions, tools, APIs and databases. 73

***REPL* – Read-Eval-Print Loop**: A read-eval-print loop is an interactive computer programming environment that takes single user inputs, evaluates them, and returns the result to the user. 18

***SBOM* – Software Bill of Materials**: The software bill of materials is a comprehensive inventory of all components, including libraries, dependencies and versions, that constitute a software product, used for tracking and managing software supply chain security. 18, 38, 73

***SE* – Software Engineering**: Software Engineering is a computing discipline. It is the systematic application of engineering approaches to the development of software. 16, 17, 18, 19, 20, 25, 29, 37, 40, 46, 47, 50, 52, 53, 66, 68, 71, 73, 74, 76

***SHA-1* – Secure Hash Algorithm 1**: SHA-1 is a hash function which takes an input and produces a 160-bit (20-byte) hash value known as a message digest – typically rendered as 40 hexadecimal digits. It was designed by the United States National Security Agency (NSA), and is a U.S. Federal Information Processing Standard. 35

***SHA-2* – Secure Hash Algorithm 2**: SHA-2 is a set of cryptographic hash functions designed by the United States National Security Agency (NSA). It consists of six hash functions with digests (hash values) that are 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256. 35

***SPDX* – Software Package Data Exchange**: The Software Package Data Exchange [13] format, created and maintained by the Linux Foundation, is a standardised way of documenting and communicating the components, licenses, and copyrights of software packages. It provides a consistent method for tracking and sharing information about software contents, particularly in open-source and collaborative environments. 38

***SRI* – Subresource Integrity**: Subresource Integrity [14] is a security feature that allows web developers to ensure that resources they fetch are delivered without unexpected manipulation. 53

***SVG* – Scalable Vector Graphics**: SVG is an XML-based vector image format. 9, 61

***SWHID* – Software Heritage Identifier**: The Software Heritage Identifier [15] is a unique identifier for software artifacts, such as source code, that is used to track and reference software across different platforms and systems. 53, 73

***SemVer* – Semantic Versioning**: Semantic Versioning [16] is a versioning scheme for software that uses a three-part version number: `MAJOR.MINOR.PATCH`. 42

***URL* – Uniform Resource Locator**: A URL is a reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it. 11

This page is intentionally left blank.

Contents

I Introduction **15**
1.1. History 16
1.2. Background 17
1.3. Motivation 19
1.4. Goals 20
1.5. Structure 20

II Reproducibility **21**
2.1. Reproducibility in Science 22
2.1.1. Reproducibility Levels 23
2.1.2. Formalisation 24
2.2. Reproducibility in Computer Science 24
2.2.1. Scope 25
2.2.2. Quantifying Reproducibility 27
2.2.3. Open Source 28
2.2.4. Terminology 29
2.2.5. Software Security 37
2.2.6. Reproducibility Utopia 40
2.3. Deterministic Builds And Environments 41
2.3.1. Sources Of Non-Determinism 41
2.3.2. Comparing Builds 45
2.3.3. Fixing Builds 46
2.4. Conclusion 46

III Software evaluation **49**
3.1. Methodology 50
3.1.1. Evaluation Criteria 51
3.1.2. Tools And Technologies 51
3.1.3. Scenarios 51
3.1.4. Compilation And Execution 51
3.1.5. Environment Setup 52
3.1.6. Output Comparison 52
3.1.7. Expected Outcomes 53
3.2. Evaluation 1 - Bare compilation 53
3.3. Evaluation 2 - Docker 54
3.4. Evaluation 3 - Guix 56
3.5. Evaluation 4 - Nix 58
3.5.1. Nix legacy method 58
3.5.2. Nix Flake 59
3.5.6. Dealing With Variability 61
3.6. Conclusion 65

IV Conclusion **67**
4.1. Summary 68
4.2. Evaluation Of Tools 68
4.3. Research Findings 72
4.3.1. At A Glance 72
4.3.2. Limitations 72
4.4. Future Work 73
4.4.1. Flaky tests 74
4.4.2. Formal Concepts 75
4.4.3. Ethical Considerations 75
4.4.4. Philosophical Considerations 75
4.4.5. Economical Considerations 76
4.4.6. Educational Considerations 76

List of definitions **79**

List of figures **81**

List of tables **83**

Bibliography **85**

This page is intentionally left blank.

Chapter I

Introduction

Sine experientia nihil sufficienter sciri potest

— R. Bacon [17]

1.1. History

R. Bacon [17], an English philosopher and scientist, articulated in 1267 the foundations of what we today recognise as "reproducibility". He famously stated in Latin "Sine experientia nihil sufficienter sciri potest" which means "Without experience nothing can be sufficiently known" [17, p.583]. He was among the first to underscore the significance of repeated experimentation as a means to test and ultimately confirm scientific findings. Although the specific term "reproducibility" was not used in his time, his emphasis on empirical evidence is seen as a precursor to our modern understanding of reproducible research in the scientific method.

Centuries later, K. R. Popper [18], an Austrian-British philosopher wrote a book on the importance of falsifiability in the scientific method. He argued that a scientific theory must be falsifiable in order to be considered valid. He also introduced the concept of *falsificationism*, which states that a theory can only be considered scientific if it is possible to conceive of an observation or an argument which proves the theory false. This principle is now widely accepted as a fundamental tenet of the scientific method.

K. Thompson [19] delivered a lecture "Reflections on Trusting Trust" at the Association for Computing Machinery (ACM) Turing Award Banquet. Since his talk, the landscape of software has undergone a radical metamorphosis. The simplicity of his advice to *trust people* has become significantly more complex in the current era. The modern software supply chain is extensive, often encompassing dependencies that lies beneath the surface. Despite the prevalence of open-source software within this supply chain, it is uncommon for end-users to compile their own software. As a result, the build systems became a prime vector for malicious exploitation [20, p.1]. This underscores the vital importance of software reproducibility which ensures that software can be reliably built and verified from its source across different environments and over time, mitigating the risks associated with trust and the potential for exploitation within the software supply chain.

J. F. Claerbout and M. Karrenbach [21] wrote about the challenges and implications of reproducibility in the paper titled "Electronic documents give reproducible research a new meaning". This work was in the field of geophysics, but it has been influential across multiple domains of Computer Science (CS). It was one of the early works to emphasise that the sharing of software and environments is critical to the reproducibility of computational results.

C. Collberg and T. A. Proebsting [22] has underscored the importance of reproducibility in Software Engineering (SE) by advocating for and funding initiatives that enhance both repeatability and reproducibility. They promote the adoption of standardised practices and transparency in research, which are crucial for ensuring that experiments are repeatable, a foundational aspect of reproducibility. By encouraging comprehensive documentation and public sharing of methodologies, data, and code, they facilitate the replication of work by others, thus enhancing the overall reliability and verification of scientific findings. Their financial support extends to tools and infrastructure that assist in establishing repeatability, which is paramount to achieving reproducibility in broader research. Recognising and rewarding efforts to share research artefacts further embeds a culture where both repeatability and reproducibility are fundamental practices in SE, ensuring that studies can not only be repeated under the same conditions but also reproduced and validated in different contexts.

In 2020, the United States sustained a sophisticated cyberattack known as SolarWinds. This meticulously orchestrated campaign apparently attributed to Russia persisted undetected for several months and was enabled through a backdoor embedded within one of the dependencies of the SolarWinds Orion (R. Alkhadra, J. Abuzaid, M. AlShammari, and N. Mohammad [23]) software, a network monitoring tool employed by numerous corporations and government agencies. By exploiting this vulnerability, the attackers gained unauthorised access to the networks of SolarWinds clients, which allowed them to steal data and deploy malware.

This incident raises a question "How did such a consequential security breach remain unnoticed for months?" and underscores the critical importance of reproducibility in SE. Had the SolarWinds Orion software and its dependencies been subject to stringent reproducibility standards, where every build could be precisely recreated and examined, the malicious alterations might have been detected earlier. Reproducibility in this context not only refers to the ability to replicate software builds but also to verify the integrity and provenance of every component, ensuring that no unauthorised changes have been

made. By prioritising reproducibility, developers and users of software can enhance security measures, mitigate risks, and foster a more trustworthy digital environment.

In a post from the Joinup - Open Source Observatory [24], institutions like the European Commission and the European Parliament began recommending Signal [25], a secure open-source instant messaging platform, for communications. This highlighted a broader, critical issue: the verification of software authenticity. Users generally trust software obtained from various stores, but this trust brings to the forefront the question of how they can verify with certainty that the version of the Signal application installed on their devices indeed derives directly from the source code provided in its repository. Ensuring an application's authenticity, confirming it has not been tampered with prior to its public release, has become a central concern in our everyday lives. This issue underscores the importance of transparent and reproducible builds where the end product can be reliably traced back to its original source, ensuring the integrity and security of the software being used.

During the OpenAI DevDay's keynote [26] in November 2023, OpenAI's CEO Sam Altman unveiled a groundbreaking feature called *reproducible outputs*. This innovation enables users to consistently replicate the outcomes generated by OpenAI's models, marking a significant advancement in achieving reproducibility within the realm of artificial intelligence.

Software companies have increasingly recognised the importance of reproducibility for enhancing security. Popular messaging platforms like Signal [27] and Telegram [28] have taken significant steps to ensure the reproducibility of their builds. They support reproducible builds, allowing users to verify that the open-source code matches the applications available on various app stores, including the Apple Store and Google Play. This initiative ensures that the distributed binaries are authentic and unaltered, thereby protecting users from potential vulnerabilities and enhancing trust in the software's integrity. By implementing these procedures, software companies highlight the broader industry's move towards transparency and reliability in software distribution, reinforcing the essential role of reproducibility in modern SE.

In the light of these considerations, this master thesis addresses the aforementioned issues and questions by delving into the principles of software reproducibility. We will explore the mechanisms that can ensure the integrity of software, and examine how these practices can be standardised to safeguard against the risks of unauthorised tampering. In doing so, this thesis aims to contribute to the critical discourse on software security and reliability in an era where digital trust is paramount.

1.2. Background

Curiosity has always been at the core of my being, fuelling an insatiable eagerness to learn and explore the unknown. I remember myself as an inquisitive boy, constantly delving into the mechanics of how things work. This curiosity often led to disassembling devices to uncover their hidden secrets, followed by a harder quest to reassemble them. While my father played a pivotal role in this journey of discovery, and I simply cannot recall a single moment when he responded with *"I don't know"* it was my mother who truly ignited my path to computers. Her encouragement and unwavering belief in pursuing my passions were instrumental in shaping my journey towards a deeper understanding of the world around me, especially computers.

My interest in computers and software development was sparked in my early childhood, before the age of ten with the Logo language, which soon led to my discovery of the BASIC programming language on an Atari 1040STE. The capabilities of that machine captivated my young mind, igniting a passion for technology and its boundless potential. This early fascination was a signpost towards my future; it was clear that my career would be intertwined with computers. Over the years, I witnessed the remarkable evolution of the software industry and the advent of groundbreaking technologies. I have also observed the progression in software development methodologies. However, despite the influx of new technologies arising, there are certain categories of issues that remain constant along the passing years, sadly.

Transitioning from BASIC, I briefly jumped on Microsoft Windows before moving to Linux, a platform that has since become my daily driver. In 2019, I found myself grappling with a sluggish laptop running a popular Linux distribution. In search of a faster, binary-based alternative, I transitioned to NixOS [29]. This shift marked the beginning of a totally new perspective on software development for me. It

was through NixOS that I encountered the concept of "reproducibility" which opened my eyes to the possibilities of making and shipping more reliable software.

We've seen in the previous section and will detail further in the next chapters that this concept originates from researchers and the scientific method. This concept can be transposed to CS and more specifically to software development. In this context, reproducibility is the ability to recreate the exact same software, including the operating system, the compiler, the libraries, and the source code, in order to obtain the same results.

For the past three years, the principle of reproducibility has totally revolutionised my approach to software development. This concept has captivated me to such an extent that I now devote a significant portion of my free time to contributing to open-source projects that emphasise reproducible builds. It is this profound interest that has inspired me to dedicate my master's thesis to exploring the depths and implications of reproducibility in SE.

Here's a non-exhausive list of projects I have contributed to:

- In the Linux NixOS operating system:
  - I created around 430 pull requests [30].
  - I made around 1800 reviews [31].
  - After several months of dedicated effort, I developed a wrapper for building reproducible Composer-based PHP applications [32], resolving a significant obstacle and positioning Nix as the preferred distribution for self-hosting PHP applications. An updated version is in preparation [33], which will be more user-friendly and will provide a more comprehensive solution for PHP developers while being at least twice faster than the previous version.
- In the PHP [34] community:
  - In Composer [35], the PHP package manager, I proposed a pull request enabling deterministic outputs by default [36].
  - I advocate for reproducibility by giving talks [37].
  - I open issues in PHP projects that are not shipping required files to enable reproducibility, explaining the reasons why it should be included:
    - In PHPUnit, the PHP testing framework [38];
    - In PsySH, a PHP REPL [39];
    - In GrumPHP, a code quality tool [40];
    - In Psalm, a static analysis tool [41];
    - In PHPMD, a static analysis tool [42];
    - In PHP-CS-Fixer, a code formatter [43];
    - In PHP-Parallel-Lint, a code linter [44];
  - I initiated and participated in discussions to improve reproducibility in the PHP source code [45].
- In the Reproducible Builds [46] project:
  - I contributed to the website by making it reproducible [47];
  - I improved the documentation [48], [49];
  - I contributed to the monthly reports [50];
- In the Typst [51] project:
  - I raised awareness about the importance of reproducibility [52];
  - I engaged in discussions on Discord leading to improving the compilation environment hermeticity [53];
- At work, I advocate the cause of reproducibility, emphasising its critical importance in our projects. The objective is to raise awareness amongst my colleagues about the advantages of reproducibility, with the ultimate aim of establishing it as a norm within our organisational software practices. As an initial measure, I am developing proofs of concept that illustrate the process of creating reproducible containers, embedding their Software Bill of Materials (SBOM) within their metadata. Additionally, I pioneered a project focused on generating ephemeral, reproducible, and tailor-made development environments and user profiles [54]. Finally, I try to provide reproducible development environments for each open-source projects [55] we publish to foster a more transparent and secure software development process but also to encourage contributions.

- In 2022, I participated in the *Summer of Nix*, a paid summer program designed to foster learning, networking, and collaboration within the Nix community. The program caters to both experienced Nix users and newcomers, offering a unique opportunity to work together on a diverse range of topics. During this event, I did a talk about how we use Nix at work [56].
- In a recent YouTube interview on "La Tronche En Biais" [57] , in a live titled "SCIENCES: Une crise de reproductibilité des études?" [58], I briefly discussed the reproducibility crisis in scientific studies and drew parallels with SE. I shared insights from my master's thesis on this topic, particularly highlighting the challenges faced when software compiled in one environment fails in another, highlighting broader implications for security and consistency across different systems. I also clarified different levels of reproducibility introduced in Chapter 2, section 1.1.

1.3. Motivation

The pursuit of reproducibility in SE is driven by a fundamental quest for precision, reliability, and trust in the digital landscape. In an age where software pervades nearly every aspect of our personal and professional lives, the importance of being able to reliably replicate software builds cannot be overstated.

At its core, software reproducibility addresses a simple yet profound question: can we consistently recreate the same software product, with the same functionality and performance, across different environments and over time? This question is not just academic but is deeply rooted in practical necessities and ethical considerations in the field of CS, but not only.

The principle of reproducibility is essential across various disciplines, ranging from the empirical rigor of scientific experiments to painting or even culinary art. This concept, at its core, is about the ability to consistently replicate results under similar conditions.

In cooking, recipes passed down through generations serve as blueprints for recreating cherished family dishes. Despite meticulously following these recipes, achieving the exact taste and texture of an ancestral meal can sometimes be elusive. Factors like cooking temperature, ingredient quality, or even altitude can possibly alter the outcome. This uncertainty in replicating results underscores the complexity and nuances involved in the process of reproducibility.

In CS, the implications of reproducibility take on a more systemic and critical dimension. Imagine wanting to use or build a software application, ensuring it is identical to what the original developer intended. For instance, this thesis itself is a digital artefact, a Portable Document Format (PDF) document derived from source files hosted in a public repository. A pertinent question arises: how can one be sure that the document produced from the source code today will be identical to one compiled a year from now? Ensuring reproducibility in such cases is not just a matter of convenience but a cornerstone for verifying authenticity and integrity in a digital world increasingly prone to misinformation and security threats.

Reproducibility in SE lies in its potential to enhance reliability and security. Reproducibility aims to eliminate the all-too-common refrain of *"it works on my machine!"* by establishing a more robust, consistent build and deployment process. It is about creating a development environment where software, when operational on one machine, can be expected to be built and function identically on another, thereby bridging gaps in consistency and predictability. In a digital era where trust and security are paramount, reproducibility is not merely a technical objective; it is a fundamental criterion for building and maintaining digital trust.

One of the driving factors behind this research is the inherent complexity present in modern software environments. Today's software systems are built on intricate layers of dependencies, including various libraries, frameworks, and operating systems. This complexity poses significant challenges in ensuring consistent behaviour of software products across different environments. Moreover, the escalating frequency of security breaches and malicious attacks on software supply chains underscores the critical role of reproducibility. It serves as a vital mechanism for verifying the integrity of software, assuring that it has not been compromised, and maintaining the transparency of the build process.

Furthermore, the academic and scientific rigors of CS demand a steadfast commitment to reproducibility. In a field where building upon previous work is the norm, the ability to validate and replicate research findings is indispensable. This aspect is particularly crucial in open-source software development, which thrives on community collaboration. The open-source paradigm hinges on the capability of developers around the world to replicate, modify, and contribute to codebases consistently and efficiently.

Lastly, the evolving nature of software poses its own set of challenges. Software development is a dynamic process, with systems continually evolving and adapting. Maintaining reproducibility ensures that earlier versions of software can be reliably reconstructed and understood, a critical factor for long-term maintenance, auditing, and compliance.

Through this thesis, the aim is to shed light on the significance of software reproducibility, exploring how it can be effectively achieved and the tools and practices that can facilitate this goal. This exploration is not only crucial for the technological advancement but also for upholding the principles of reliability, security, and transparency in an increasingly software-dependent world.

1.4. Goals

In this master thesis, my primary focus will be to provide a comprehensive overview of reproducible builds, within the sphere of software development, acknowledging that a complete examination of every aspect of reproducibility is beyond our scope.

I will explore a selection of tools and methodologies that promote reproducibility or, at least, create favorable conditions for facilitating it. Moreover, this document is intended to enlighten and hopefully convince the reader that the construction of reproducible software should be a fundamental principle, not merely a secondary consideration, within the software development lifecycle. Finally, I will delve into the rationale for adopting this reproducibility paradigm as a standard practice in modern SE, with particular emphasis on security implications.

By the conclusion of this thesis, the reader will have a comprehensive understanding of the concept of reproducibility and how best practices can be implemented effectively in software development projects.

1.5. Structure

Organised into several chapters, this thesis systematically explore the multifaceted nature of software reproducibility.

- Chapter 1 being this introduction, outlining the thesis's scope and objectives.
- Chapter 2 introduces the origin of the concept of reproducibility, tracing its lineage from scientific principles. It proposes a terminology, formalisms and its challenges.
- Chapter 3 is a hands-on exploration, delving into specific real-world examples. It will demonstrate practical applications of the concepts discussed in previous chapters, including proof of concept implementations, concrete case studies, and detailed analyses of real-world scenarios where reproducibility plays a crucial role. This chapter aims to bridge theory with practice, showing how the principles of reproducibility are applied and sometimes challenged in real-world settings.
- The final Chapter 4 synthesises the insights gained throughout the thesis. It offers recommendations for best practices based on the research and discussions presented. Moreover, it suggests directions for future work, identifying areas where further research, development, and discussion are needed to advance the field of software reproducibility.

Chapter II

Reproducibility

Reproducibility is a minimum necessary condition for a finding to be believable and informative.

— J. T. Cacioppo, R. M. Kaplan, J. A. Krosnick, J. L. Olds, and H. Dean [59]

2.1. Reproducibility in Science

> "*No serious physicist would offer for publication, as a scientific discovery, one for whose reproduction he could give no instructions.*"
>
> K. R. Popper [18]

The concept of reproducibility lies at the heart of scientific inquiry, serving as a critical benchmark for the validation and acceptance of research findings. It is a principle that transcends scientific disciplines, insisting that the results of an experiment or study must be consistently replicable under identical conditions by different researchers. This aspect of the scientific method ensures the reliability and integrity of scientific knowledge. It establishes a framework where hypotheses are not just tested but also subjected to repeated verification, underpinning the trust and credibility that society places in scientific discoveries. The journey of reproducibility, originating from the earliest scientific endeavors, has evolved to adapt to the complexities and nuances of modern research methodologies. This evolution mirrors the progression of scientific thought and technology, from rudimentary experiments to sophisticated, computer-assisted analyses.

One can observe the glimpse of the first traces of this concept in K. R. Popper [18]. The concept of reproducibility is far from new and has been a cornerstone in the sciences for centuries. It aims to explain natural phenomena in an objective and repeatable manner.

According to M. Castillo [60], the scientific method (Figure 1), a formalised and widely-adopted process for exploring observations and answering questions, is inherently designed to be repeatable. However, this does not guarantee that the results of all experiments conducted using the scientific method will be reproducible. When results cannot be replicated, it raises questions about the validity of the experiment and the credibility of the researcher.

In the realm of scientific research, *repeatable* and *reproducible* are terms often used interchangeably, yet they hold distinct meanings.

Repeatable research refers to the ability of a study or experiment to yield the same results when conducted again under the same conditions with the same materials and methods by the same researchers. It primarily focuses on the consistency and reliability of results within the original research context.

On the other hand, *reproducible research* emphasises the ability of an independent researcher to attain at the same findings and conclusions using the original study's raw data and following the same methodologies, but possibly under different conditions and with different tools. Reproducibility extends the validation process beyond the original researchers, ensuring that the results hold up under scrutiny and can be reliably used as a foundation for further study.

Together, repeatability and reproducibility are foundational to the integrity and advancement of scientific knowledge, allowing for a deeper trust and understanding of research findings.

While reproducibility can be considered closely aligned with the scientific method, it is not an intrinsic part of it. The scientific method is a procedural approach for conducting experiments, whereas reproducibility is a quality attribute of the experimental results (Figure 2).

As of 2016, some of its basic terms were not standardised. This diverse nomenclature has led to confusion, both conceptual and operational, about what kind of confirmation is needed to trust a given scientific result.

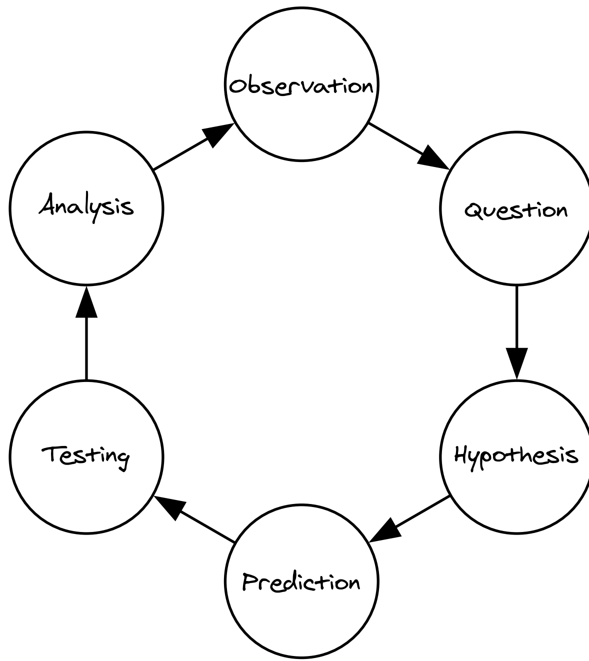


Figure 1: Scientific method

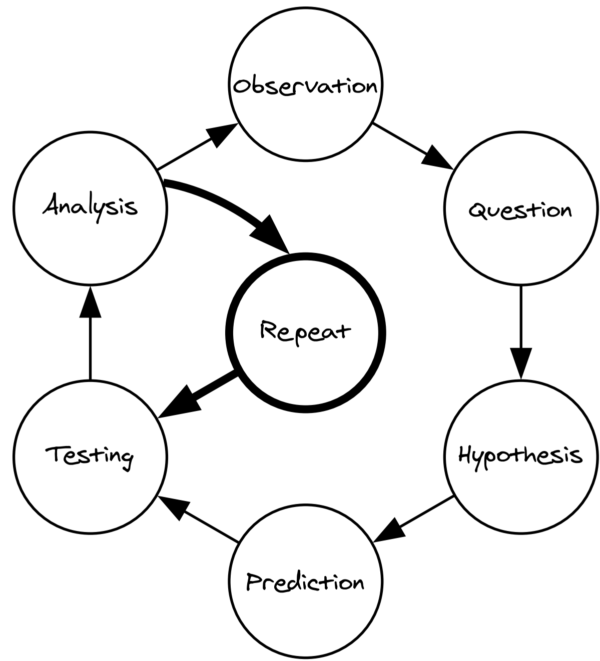


Figure 2: Scientific method with reproducibility

Reproducibility in research is a major factor that determines the uniqueness of research studies. It means obtaining consistent results using the same data and protocol as the original study. For example, researchers confirm the validity of a new discovery by repeating the experiments that produced the original results. Moreover, other researchers in the field are also able to repeat the same experiments producing the results similar to the original.

2.1.1. Reproducibility Levels

According to B. T. Essawy *et al.* [61], reproducibility is organised in four levels:

- **Repeatability**: Achieved upon obtaining consistent results using the same input data, computational steps, methods, and code on the original researcher's machine. This level is normally achieved in scientific papers.
- **Runnability**: Achieved when the author of the research can obtain consistent results using the same input data, computational steps, methods, code and conditions of analysis on a new machine.
- **Reproducibility**: Achieved when a new researcher, not an original author of the analysis, is able to reproduce the analysis in their own computational environment [62].
- **Replicability**: Achieved by obtaining consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data [62]. Replicability also allows scientists not involved in the original study to build from and expand on research once they are first able to reproduce that research.

Table 1: The four levels of reproducibility and their requirements.

	Original			Other		
	researcher	**machine**	**data**	**researcher**	**machine**	**data**
Repeatability	✓	✓	✓			
Runnability	✓		✓		✓	
Reproducibility			✓	✓	✓	
Replicability				✓	✓	✓

It's crucial to understand that these levels are interconnected and not isolated. Achieving reproducibility level means that the criteria for both repeatability and runnability levels have been met.

Table 1 delineates four levels of reproducibility, each with specific prerequisites. It's important to acknowledge that these levels are organised in ascending order of difficulty to attain, starting from the simplest to the most challenging. Consequently, progressing through these levels necessitates an incremental investment of resources, time, and effort.

2.1.2. Formalisation

DEFINITION 1: EXPERIMENT

An experiment e conducted with a set of parameters and conditions p where $r(e, p)$ represents the experiment results, is *reproducible* if and only if:

$$\forall e \in E, \forall e' \in R(e), \forall p \in \operatorname{par}(e), \quad r(e, p) = r(e', p)$$

where

- E is the set of all possible experiments
- par is a function defined as $\operatorname{par} : E \to \mathcal{P}(P)$ where $\mathcal{P}(P)$ is the powerset of P, the set of all possible parameters of all possible experiments
- $R(e)$ is a function defined as $R : E \to \mathcal{P}(E)$ where $\mathcal{P}(E)$ is a powerset of E that gives for each experiment $e \in E$, its set of independent replications

2.2. Reproducibility in Computer Science

"*In their vision of reproducible research, readers should be able to rebuild published results using the author's underlying programs and raw data. Implicitly, they are advocating for open code and data.*"

L. A. Barba [63]

As we shift our focus from general scientific domains to the realm of CS, the principles of reproducibility undergo a unique transformation. In our digital era, where computations and algorithms form the backbone of research, reproducibility challenges and solutions take on new dimensions. The intricate interplay of software, hardware, and data in CS demands a re-examination and adaptation of traditional reproducibility concepts. This is where the principles established in the broader scientific community intersect with the specificities of computing, leading to a distinct and crucial discourse on reproducibility in the field of CS.

The initial recorded use of the term *reproducible research* in an academic paper is believed to have occurred in 1992, in a presentation by J. F. Claerbout and M. Karrenbach [21]'s team at Stanford, during the Society of Exploration Geophysics conference. M. Schwab, N. Karrenbach, and J. Claerbout [64], the same group of researchers updated their definition of *reproducibility in computationally oriented research*.

In D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden [65], it is stated that reproducibility depends on open code and data. The authors define reproducible computational research as that *in which all details of computations, code and data, are made conveniently available to others.*

DEFINITION 2: REPEATABILITY (SAME TEAM, SAME EXPERIMENTAL SETUP)

The measurement can be obtained with stated precision by the same team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same location on multiple trials. For computational experiments, this means that a researcher can reliably repeat her own computation.

S. N. Goodman, D. Fanelli, and J. P. A. Ioannidis [66] acknowledge the lack of standardisation in foundational terms like reproducibility, replicability, reliability, robustness, and generalizability. To address this, they suggest a new lexicon: *Methods Reproducibility* to align with the original concept of reproducibility as defined by J. F. Claerbout and M. Karrenbach [21] and D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden [65], *Results Reproducibility* corresponding to what R. D. Peng [67] refers to as replicability, and *Inferential Reproducibility* to denote a distinct category.

Definition 3: Reproducibility (Different team, different experimental setup)

The measurement can be obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using the author's own artefacts.

The term *reproducibility* in the context of CS has been refined and explored in many subsequent works and identifying a single *first* definition can be challenging due to the evolution of the concept over times. According to L. A. Barba [63], who conducted a detailed article on the terminology history, the most appropriate terminology (Definition 2, Definition 3, Definition 4) to describe reproducibility in the context of CS is the definitions derived from Association for Computing Machinery [68, Artifact Review and Badging].

Definition 4: Replicability (Different team, same experimental setup)

The measurement can be obtained with stated precision by a different team, a different measuring system, in a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using artefacts which they develop completely independently.

In the context of this document, Definition 5 is the definition of *reproducibility* that we will use when referring to the concept of reproducibility in CS.

Definition 5: Reproducibility

Reproducibility is the ability to consistently obtain identical results across multiple runs of a computer task when using the same methods and data, regardless of method, space and time. Note that this does not necessarily imply that the outputs are correct or the desired outputs.

Space and *Time* are terms borrowed from physics. In the context of reproducibility in SE, space refers to different systems, while time refers to different moments in time [69]. (more about that in Definition 14).

2.2.1. Scope

In this master thesis, the exploration of reproducibility will focus on a specific aspect: ensuring the reproducibility of building source code to ensure that the resulting application works. This critical area is paramount in the field of SE, where the ability to consistently recreate identical software artefacts from a given source code under varying conditions and environments stands as a paramount concern. The intent is not to undermine the importance of these other facets, but rather to mainly concentrate the efforts and analysis on the reproducibility of source code building and compilation.

We acknowledge that languages like JavaScript, PHP, Python are not compiled but merely interpreted by their respective interpreter. Often, these scripting languages require dependencies provided by their respective package manager as well. Ensuring the availability of these dependencies is an integral part of the software build process and, to some extent, corresponds to the compilation in non-compiled languages. Consequently, in the context of this thesis, "compiling source code" is applicable to both types of programming languages.

The concept of reproducibility can be distinctly categorised into multiples phases like: *reproducibility at build time* (Definition 6) and *reproducibility at run time* (Definition 7). It is important to note that these phases are not mutually exclusive and can be combined to achieve a higher level of reproducibility.

Definition 6: Reproducibility at build time

Reproducibility at build time refers to the ability to consistently generate the same executable or software artefact from a given source code across different builds on different environments, across different space and time. This aspect is crucial in ensuring that the software compilation process is deterministic and immune to variances in development environments, compiler versions, or build tools. It involves a meticulous standardisation and documentation of the build environment and dependencies to guarantee that the same executable is produced regardless of when or where the build occurs.

Definition 7: Reproducibility at run time

Reproducibility at run time addresses the consistency of software behaviour and output when the software is executed in different environments or under varying conditions. This type of reproducibility focuses on ensuring that the software performs identically and produces the same results regardless of the Operating System (OS), underlying hardware, or external dependencies it interacts with during execution.

To illustrate these phases, the C source code in Listing 1 implements the Monte Carlo method to approximate the value of π. This is an example of reproducibility at build time, but not at run time.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char* argv[]) {
  srand(time(NULL));

  double x,y,z,count;
  int n = atoi(argv[1]);

  for (int i = 0; i < n; i++) {
    x = (double) rand() / RAND_MAX;
    y = (double) rand() / RAND_MAX;
    z = x * x + y * y;
    if (z <= 1) count++;
  }

  printf("π approx.: %g", (count / n) * 4);

  return(0);
}
```

Listing 1: `montecarlo-pi.c`

```
$ gcc montecarlo-pi.c -o montecarlo-pi
$ sha256sum montecarlo-pi
241d0d05314472c6472fd90814f253ccc1b4fd10366ab828201aab1ac1a0d376  montecarlo-pi
$ rm montecarlo-pi
$ gcc montecarlo-pi.c -o montecarlo-pi
$ sha256sum montecarlo-pi
241d0d05314472c6472fd90814f253ccc1b4fd10366ab828201aab1ac1a0d376  montecarlo-pi
```

Terminal session 1: Building the same source code multiple times always yields the same binary executable

The Monte Carlo algorithm is inherently stochastic, it uses random sampling or probabilistic simulation as a core part of its computation. This randomness is intrisic to the algorithm's design and purpose.

The distinction between build time and run time reproducibility for the Monte Carlo algorithm arises from its usage of a random source. While the algorithm can be reliably built into a consistent binary (build time reproducibility), its outputs can vary on different executions under the same conditions due to its inherent randomness (lack of runtime reproducibility). This does not undermine the validity of the algorithm but rather is a characteristic of its probabilistic approach to problem-solving.

```
$ gcc montecarlo-pi.c -o montecarlo-pi
$ ./montecarlo-pi 10
3.6
$ ./montecarlo-pi 10
4
$ ./montecarlo-pi 1000000
3.13989
$ ./montecarlo-pi 1000000
3.14048
```

Terminal session 2: Running the binary multiple times does not always yields the same result

In practice, for certain applications, runtime reproducibility can be attained by controlling the random number generator, specifically by setting a fixed seed as an input parameter.

In the next example, the source code is not reproducible at build time and we might erroneously think that the program is reproducible at run time.

We observed that compiling the same source code multiple times results in different binaries. This variation occurs because the source code includes the macros `__TIME__` and `__DATE__`, which are substituted with the current time and date during compilation. As a result, we cannot achieve reproducibility at build time.

```c
#include <stdio.h>

int main() {
  printf(
    "Built the %s at %s.\n",
    __DATE__,
    __TIME__
  );
  return 0;
}
```

Listing 2: Sourcecode of `datetime.c`, a C program with macros

Upon executing the produced binary, the outcome appears consistent. This might suggest that the binary is reproducible at run time, however, this assumption is incorrect. Consider a scenario where a different user compiles the same source code. In such a case, runtime reproducibility between the original and another user is not assured.

```
$ gcc datetime.c -o datetime
$ sha256sum datetime
a123b...c8dba  datetime
$ rm datetime
$ gcc datetime.c -o datetime
$ sha256sum datetime
937da...1284f datetime
$ ./datetime
Built the Nov 21 2023 at 17:19:34.
$ ./datetime
Built the Nov 21 2023 at 17:19:34.
```

Terminal session 3: An example of program that it neither reproducible at build time and run time.

2.2.2. Quantifying Reproducibility

Quantifying reproducibility is traditionally viewed as a binary state: it is either reproducible or not. However, this perspective oversimplifies the complexity of software environments. In reality, reproducibility exists on a spectrum, where the focus shifts from a mere *yes-or-no* assessment to evaluating the extent and conditions under which a computation is reproducible.

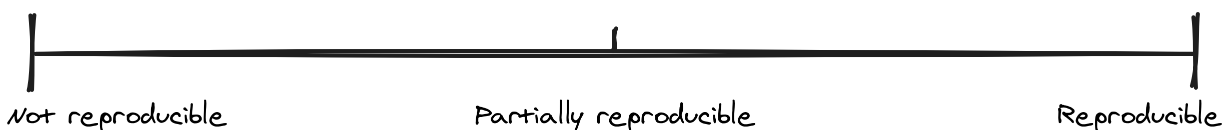


Figure 3: Reproducibility states

We will explore this concept with Docker images as a primary example. Docker, a popular containerization platform, uses Dockerfiles (Listing 3). Basically, a `Dockerfile` is a script with a set of instructions to build images. These images are then used to run software in a consistent environment. However, many images on the Docker Hub [70] present challenges to reproducibility. The reasons vary: some Dockerfiles are not publicly available but especially because most of them include significant variability in their build processes, making exact replication of the images pretty much impossible. We will consider these challenges in more detail in Chapter 4.

```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Listing 3: An example of `Dockerfile`

Determining a precise value for a Docker image's temporal reproducibility is complex. Thus, for the purposes of this thesis, we simplify the classification into three broad categories as outlined in Figure 3: *Not reproducible*, *Partially reproducible*, *Reproducible*. While more nuanced classifications are possible, this simplified tripartite model provides a sufficient basis in this thesis.

Despite these challenges, Docker images are widely used. They remain static between updates, creating a window during which their environment is consistent. This period (the interval between two successive updates) can serve as an indirect measure of reproducibility. Essentially, the longer the time between updates, the more stable and, by extension, reproducible the image is considered. In considering the temporal dimension of reproducibility, it is essential to recognise that software artefacts are not unchanging entities; they offer a predictable environment for a finite period. Imagine a scale where 0 represents non-reproducibility and 1 indicates full reproducibility. On this scale, the temporal reproducibility of Docker images would be positioned between 0 and 1, acknowledging the nuanced nature of this concept.

However, this reproducibility is inherently dynamic due to the nature of software evolution. Each update to a Docker image might introduce changes affecting the software's behaviour, thereby impacting its reproducibility over time. Understanding this aspect of temporal reproducibility is crucial for managing software environments in a continuously advancing technological landscape.

The Docker use case can be classified under the *Partially reproducible* class. This is because, while Docker images ensure reproducibility at run time by providing a consistent execution environment, they often fall short of reproducibility at build time due to the variability inherent in their Dockerfiles. This dichotomy highlights the spectrum of reproducibility, where Docker images occupy an intermediate position. They are neither fully reproducible (due to build-time variability) nor completely irreproducible (thanks to their runtime stability). This categorization not only helps in understanding the reproducibility status of Docker images but also underscores the need for a nuanced approach to classifying software reproducibility, acknowledging the various shades that exist between the *Not reproducible* and *Reproducible* boundaries.

2.2.3. Open Source

> “*If everyone on a research team knows that everything they do is going to someday be published for reproducibility, they'll behave differently from day one. Striving for reproducibility imposes a discipline that leads to better work.*”
>
> D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden [65]

Open Source refers to a type of software whose source code is freely available for anyone to view, modify, and distribute. This model encourages collaborative development and sharing, allowing users and developers to improve the software and adapt it to their needs.

Open-source software development, known for managing complex projects with high quality, significantly enhances reproducibility by fostering professionalism and transparency. Making open-source software reproducible offers numerous advantages: it streamlines the onboarding of new contributors, improves

testing and feature implementation, ensures transparent build processes, facilitates security audits, and quickens response times in the dependency supply chain in case of issues.

Reproducibility, intrinsically linked with Open Source, is fundamentally an activity that builds trust (L. A. Barba [71]), making it a leading method for ensuring software can be reliably built and verified by a diverse global community.

A direct correspondence can be established with the taxonomy of B. T. Essawy *et al.* [61] when considering the process of building software from source code. This process can be analogised to a scientific experiment. In this context, the act of an individual building the source code of another developer on their own machine mirrors the concept of *"Reproducibility"* as described in Table 1. This signifies that the software, when compiled by different users from its source code, consistently results in the same executable or software artefact. The transparency inherent in open-source software is a foundational advantage. Since the source code is publicly available, it allows researchers to scrutinise how the software operates, understand how results are generated, and validate the reliability and accuracy of the software. This level of openness is crucial for replicability and trust in scientific research.

Furthermore, open-source software promotes a culture of collaboration and community involvement. Active communities that grow around open-source projects contribute to the software's continual improvement. This community-driven development leads to the identification and resolution of bugs, thereby enhancing the software's reliability and, consequently, the reproducibility outcomes that depend on it.

A key feature of open-source software is the permissive nature of its licencing, which, depending on the specific licence, facilitates the reuse and modification of software without legal or technical barriers. This flexibility is vital for verifying and replicating studies, as researchers can adapt the software for their specific needs without restrictions, though some licences may impose certain conditions. Additionally, open-source development tools provide excellent record-keeping capabilities, like version control systems (e.g., `git`, Mercurial, Pijul), enabling researchers to track changes and understand the context of each update. This aspect is essential for reproducing and validating research findings.

Lastly, the open source approach aligns well with the scientific values of openness and sharing, promoting a culture that values transparency and reproducibility in scientific inquiry. Moreover, the community-driven nature of open-source software reduces the risk of obsolescence, ensuring that research tools remain accessible and up-to-date for future replication efforts.

In essence, open-source software embodies a framework that is not only conducive to the scientific pursuit of knowledge but also reinforces the integrity and sustainability of SE through its emphasis on transparency, collaboration, and adaptability.

Open-source development, by its nature of allowing anyone to build, verify and use software, stands out as an effective, if not the best, approach to bolstering both confidence and safety in software systems. This widespread participation and verification process inherent in open-source development contributes significantly to the reliability and security of the software.

2.2.4. Terminology

Establishing formal definitions and terminology is crucial for aligning researchers, practitioners, and readers on the same wavelength. By articulating a clear and precise mathematical representation, we facilitate a universal understanding of what it means for a computation to be reproducible.

The following section is dedicated to constructing such a formal definitions, a balance between the rigor required by the academic community and the clarity needed for widespread adoption.

2.2.4.1. Computation

A computation is a process that involves the execution of algorithms or a series of operations to obtain a result, usually performed by a computer. It can be complex, involving multiple steps, conditions, and data manipulations (Figure 4). The formal definition of computation takes into account the computational environment variable, reflecting the specific context where the computation occurs.

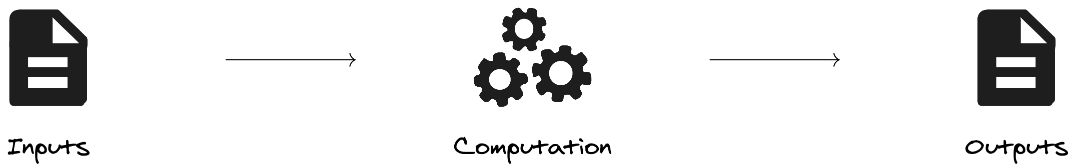


Figure 4: Inputs, computation, outputs

In the context of CS, defining a computation involves considering the broader scope of activities and processes that a computer performs, extending beyond the traditional mathematical abstraction of a function. A computation can be understood as a sequence of steps or operations performed by a computer to transform input data into output data. This process can involve various types of functions, algorithms and data manipulations. Essentially, a computation can be depicted as an abstraction involving one or multiple functions.

Examples of computations could be: a program build, a compilation, a program execution, a data analysis, a data transformation.

- When source code is compiled, the input is the source code and the output is the binary executable.
- When a program is executed, the input is the binary and the output is the result of the program.
- When making a data analysis, the input is the raw data and the output is the analysis.
- When evaluating a function, method, or procedure in any programming language, the input consists of the function itself along with its parameters. The output is the result of the function applied to these parameters, including any potential side effects (e.g., changes in the program's state).

Definition 8: Computation

A computation c is a set of one or more functions $f : I \times E \to R$.

where
- I is the set of all possible arguments or inputs the computation needs
- E is the set of all possible execution environments (hardware, software, space, time)
- R is the set of all possible outputs

It is crucial to distinguish functions, both of which are integral in the realms of Mathematics and CS. In Mathematics, a function is a deterministic construct defining a specific relationship between sets of inputs and outputs, mapping each input to exactly one output. It acts as a fundamental building block within computations to describe how values are transformed. In CS, functions are similar but can be classified as pure (Chapter 2, subsection 2.4.4) or impure (Chapter 2, subsection 2.4.5), with pure functions having no side effects and impure functions potentially affecting the state or relying on external variables. While a function provides the rules for individual transformations within a computation, the computation itself represents the broader and more dynamic process of achieving a result, often involving the execution of complex algorithms, data handling, and the application of multiple functions and operations.

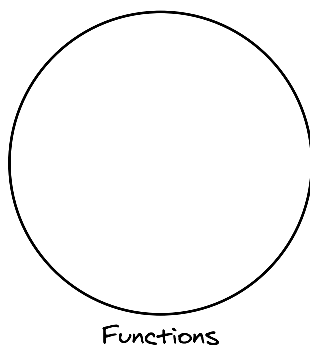


Figure 5: Functions in the context of Mathematics

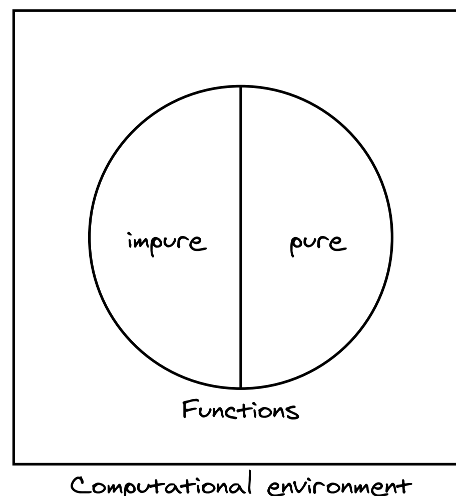


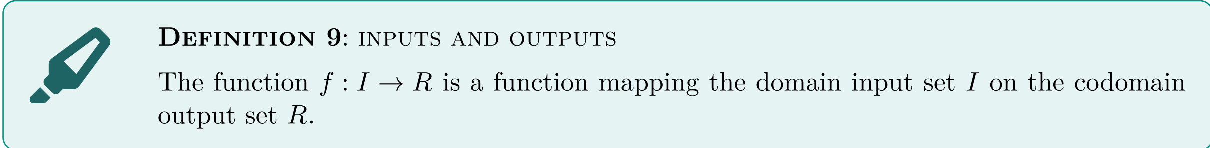
Figure 6: Functions in the context of CS

In CS (Figure 6), a function necessitates an environment in which it will be evaluated, effectively making, to some extent, this environment an extra input parameter per se. This computational environment, which encompasses the hardware (e.g., filesystem, memory, CPU), software (e.g., OS) and date (e.g., the current date and time), may influence the function's behaviour and output. Consequently, functions in CS are inherently designed to interact with and adapt to their environment, thereby making them dynamic and versatile but also potentially non-deterministic.

Conversely, in Mathematics (Figure 5), a function is evaluated independently of any environment, or with the environment variable effectively set to null, ensuring its behaviour is entirely predictable and self-contained. This means its behaviour is entirely predictable and self-contained. This distinction highlights the adaptability and complexity of functions in computational contexts, compared to their more stable and defined mathematical equivalents.

2.2.4.2. Inputs and Outputs

An input is the data provided to a computation. The output is the result of a computation or any other changes made to the environment the computation is being evaluated in.



Definition 9: inputs and outputs

The function $f : I \to R$ is a function mapping the domain input set I on the codomain output set R.

Inputs and outputs can vary widely, ranging from user interactions and network connections to files and directories. The nature of these inputs and outputs significantly impacts the reproducibility of computational processes.

Consider user interactions, such as mouse or eyes movements. These are inherently challenging to replicate precisely due to their dynamic and unpredictable nature. For instance, reproducing the exact trajectory of a mouse movement is virtually impossible due to the minute variations in human actions. However, a more reproducible approach would be to capture these interactions in a structured format like in `eyeScrollR` (N. Larigaldie, A. Dreneva, and J. L. Orquin [72]). Recording the coordinates of mouse movements over time in a file creates a detailed log that can be replayed. This arbitrary method transforms a non-reproducible user interaction into a reproducible set of data.

For a computation to be considered reproducible, its inputs and outputs must be storable and retrievable. Typically, the most feasible types for such storage are files or directories, primarily due to the ubiquity and accessibility of file systems in computing environments. Files and directories offer a stable and widely accessible medium to store and retrieve data.

In this thesis, the focus will be on scenarios where inputs and outputs are in the form of files, unless specified otherwise. This assumption aligns with the common practices in computational processes and aids in maintaining the reproducibility of the computations discussed.

In the context of software compilation, an output is correct when it faithfully reflects the state of its transitive inputs. Basically, the output represents all direct and indirect dependencies used in the build process. "Transitive inputs" refer to not only the direct inputs (e.g., source code) but also to the inputs of those inputs (e.g., libraries, frameworks, compilers, data resources).

From the point of view of the software build process as shown in Figure 7, the inputs are all the source code files, configuration files, and dependencies required to build the software.

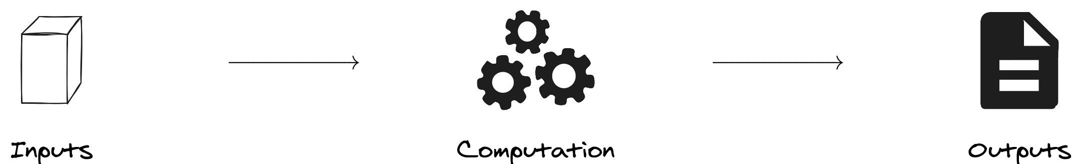


Figure 7: Inputs, computation, outputs

In Figure 8, the process has been refined from the perspective of the user running the software, where the input is now composed of the program and its parameters. This distinction is crucial as it highlights the

dynamic nature of computational processes. The user's interaction with the software, such as providing parameters or executing commands, is integral to the inputs and can significantly influence the output.

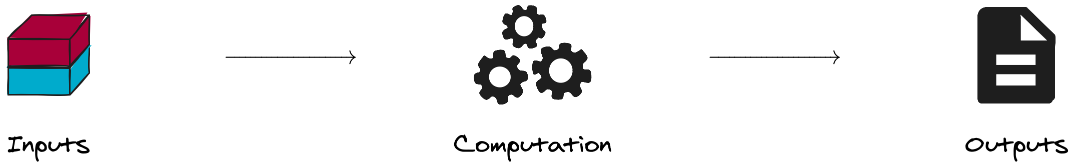


Figure 8: The input is now composed of the program and its parameters

In Figure 9, the environment where the computation is evaluated is added to the input. This environment includes the hardware, software, space, and time in which the computation is executed. This addition further refines the definition of inputs and outputs, emphasising the dynamic and context-dependent nature of computational processes.

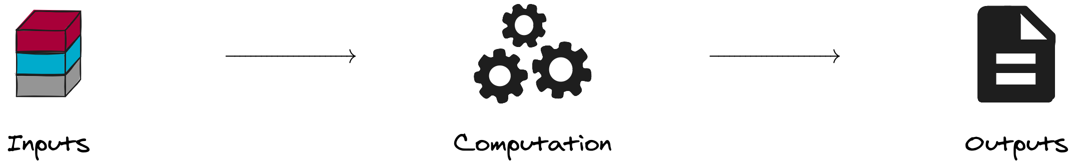


Figure 9: The input is now composed of the program and its parameters and the environment where it is going to be evalued.

We could break down the environment further. However, as we delve deeper into segmenting the components essential for a computation, the process becomes increasingly subjective (K. Hinsen [73]).

Reproducibility implies to compare outputs to determine if they are equivalent. According to P. Ivie and D. Thain [74, p.5], there are multiple equivalence classes:

Table 2: Classes of reproducibility

Equivalence class	**Examples**
Same phenomenon	Human experts
Same statistics	Software like GNUplot, Matplotlib, R
Same data	Checksum of file contents
Same bits	Checksum of file contents and metadata

- Two natural phenomena could be observed by human experts and considered as the same.
- Two results could be statistically equivalent, in that the numeric values are different, but they both convey the same statistical interpretation.
- Two results could be the same data in the sense that they encode the same numeric contents, but differ in some irrelevant detail. For example, an output file might incidentally contain the system time and the name of the user who ran the program.
- Two results could be equivalent, in every way, bit-per-bit. This is the strictest form of equivalence.

In the context of this thesis, we will assume that two results are equivalent if they are the same, bit-per-bit.

It is important to clarify that in the context of reproducibility, the time taken to compute the output is not typically considered. This means that two results can be deemed equivalent or reproducible even if the computational time to achieve these results varies. For instance, consider a situation where a piece of code is refactored: if the output data remains unchanged, the process is considered reproducible from a data consistency perspective. Nonetheless, even if the refactored code requires significantly more time and resources to execute, it is still classified as reproducible as long as the output remains consistent with the original.

This distinction underscores that reproducibility focuses on the consistency of the output data rather than the performance or efficiency of the computational process. This aspect is particularly relevant in environments where hardware or system efficiencies may differ, yet the integrity and equivalence of the output data remain the primary concern. While this might provoke debate regarding resource efficiency and computational time, for the purposes of this thesis, it is assumed that the temporal and resource aspects of computing the output are secondary to the consistency of the results themselves.

2.2.4.3. Evaluation of a Computation

The evaluation of a computation is the process of determining the resulting output of a function for a given set of arguments. It involves applying, in a specific computational environment, the function's defined operations to the inputs to produce an output. This does not necessarily imply that the outputs are correct. Note that, *correct* means that the evaluation has successfully completed without errors.

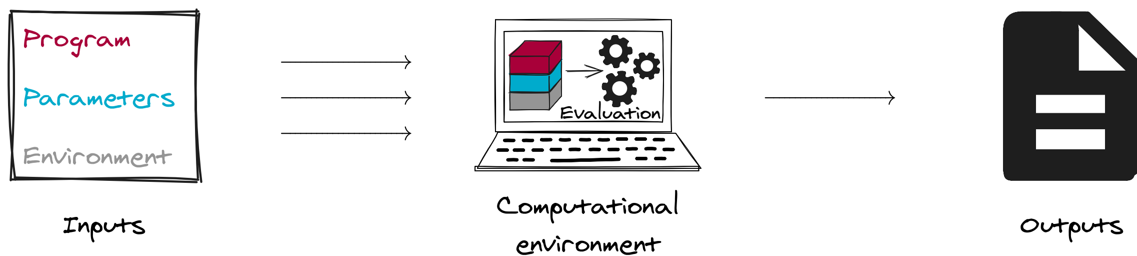


Figure 10: The evaluation of inputs into outputs where the input is composed of the program and its parameters and the environment where it is going to be evaluated.

Definition 10: Evaluation

$\text{eval} : (F, I, E) \to R$ is a function that evaluates a function f and its parameters i in a specific computational environment e to produce a result, an output.

$$\forall f \in F, \forall i \in I, \forall e \in E, \quad \text{eval}(f, i, e) = f(i, e)$$

where

- F is the set of all possible computations
- I is the set of all possible arguments the computation needs
- E is the set of all possible execution environments (hardware, software, space, time)
- R is the set of all possible outputs

In the realm of mathematics, a function is typically isolated, operating solely on its provided arguments, with no external environmental factors influencing its output. Contrarily in CS, it is quite common for a computation to interact with, and be influenced by, its surrounding environment during evaluation, which necessitates Definition 10.

2.2.4.4. Pure Function

As seen in Figure 5, the concept of a *pure function* as defined in Definition 11 does not explicitly exist in Mathematics. This is because functions are always inherently considered to be deterministic and side-effect free, here functions in maths are *by default* pure. Any mathematical function evaluates under the assumption that given the same inputs, the output will always be the same, and the evaluation of the function does not alter any external state or variable.

Definition 11: Pure function

A pure function can be defined as a function where the same input always yields to the same output.

Let $f : I \times E \to R$ be a function. Then f is **pure** if and only if:

$$\forall i \in I, \forall e_1, e_2 \in E, \quad \text{eval}(f, i, e_1) = \text{eval}(f, i, e_2)$$

where
- I is the set of all possible inputs arguments
- E is the set of all possible execution environments (hardware, software, space, time)

A bridge can be drawn between the mathematical definition of a function $f : I \to R$ and this definition by considering the environment variable E as an empty set, making the function independent of any external state or variable. This effectively reduces the definition of a pure function in CS to the mathematical definition of a function.

However in CS, it makes sense to define what are pure and impure functions because a function might behave differently depending on the environment in which it is executed. Therefore, the purity of a function in the context of CS is vital for understanding and managing side effects and state in software, it is a distinction that doesn't apply in the static, deterministic realm of pure mathematics.

This distinction highlights how the same term can have different implications in different disciplines, reflecting the unique nature of challenges and concepts in programming versus pure mathematics. However, we can still try to define such a function in a theoretical CS context.

A pure function is a specific type of function in programming characterised by the following properties:

- Deterministic: for a given set of inputs, a pure function always returns the same output. This means the function's output is solely determined by its input values and does not rely on any external state or data.
- No side effects: A pure function does not cause any observable side effects in the system. This means it does not modify any external state, global variables, or data outside its scope. It also does not produce outputs other than its return value, such as printing to the console or altering the state of the program beyond the scope of the function.

A checksum(Chapter 2, subsection 2.4.6) is an example of pure function. It will constantly return the same output for the given output.

2.2.4.5. Impure Function

An impure function is the opposite of the above definition of a pure function.

Definition 12: Impure function

An impure function is a function that does not always yields the same output for a given input. This can be formally expressed as:

Let $f : I \times E \to R$ be a function. Then f is **impure** if and only if:

$$\forall i \in I, \exists e_1, e_2 \in E, \quad \text{eval}(f, i, e_1) \neq \text{eval}(f, i, e_2)$$

where
- I is the set of all possible inputs arguments
- E is the set of all possible execution environments (hardware, software, space, time)

It is a specific type of function in programming characterised by the following properties:

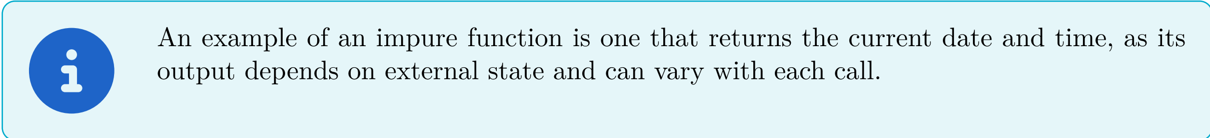
- Non-deterministic: the function can yield different outputs for the same set of input values at different times, depending on the state of the system or environment in which it is executed.

- Side effects: the function performs actions that modify some state outside its local environment or has observable interactions with the outside world. This can include altering global variables, modifying input arguments, I/O operations, or calling other impure functions.

As seen in Figure 6, this concept only exists in programming, as it is a direct consequence of the mutable nature of the state in programming. In pure mathematics, functions are conceptualised as mappings from elements of one set (the domain) to elements of another set (the codomain), without any side effects or external dependencies. This distinction highlights the difference between the theoretical framework of mathematics and the practical aspects of programming, where functions often interact with a mutable state or environment.

Given this, we will allow ourselves to define such a function in the theoretical context of computer science. Implying that to define such a function, an additional parameter, which will be used to calculate the time, *must* be passed as a parameter to the function. This parameter corresponds to the `E` parameter in Definition 12.

Given this, we will allow ourselves to define such a function within the theoretical context of CS.



An example of an impure function is one that returns the current date and time, as its output depends on external state and can vary with each call.

2.2.4.6. Checksum

Although understanding the concept of a checksum is not essential for understanding the definitions, it is crucial to define it due to its recurring presence in the next chapters.

A checksum is the result of a computation. It is a one-way pure function which takes an input of an arbitrary size and returns a string of a fixed size, depending on the checksum algorithm in use. For example when using a `git`, each commit ID is a checksum of the current commit's content and the previous commit's ID.

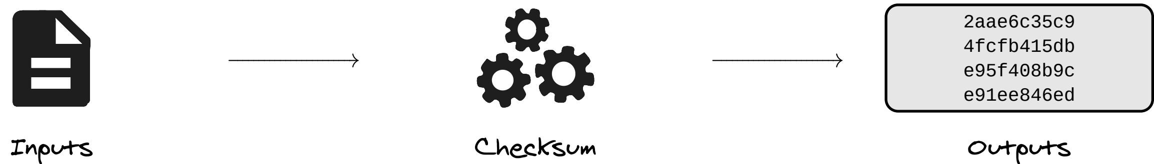


Figure 11: Inputs, checksum, string output

A one-way function is easy to compute but is practically impossible to reverse. This is mostly due to the fixed size output, the number of possible inputs (*domain*) exceeds the number of possible outputs (*codomain*). The time complexity of such a function is usually linear, which means that the time it takes to compute the checksum is proportional to the size of the input, therefore $\mathcal{O}(n)$.

A checksum is a function that returns a string called *hash*, which is supposedly unique for a given input. Checksum algorithms are designed to produce a unique hash for each unique input. However, the term "supposedly unique" is used because, in theory, it is possible for two different inputs to produce the same hash, an occurrence known as a *collision*. The ability to find collisions undermines the security of the algorithm. There are different types of algorithms to calculate a checksum (e.g., MD5, SHA-1, SHA-2). Older algorithms like MD5 have known vulnerabilities that allow collision attacks while more modern algorithms like SHA-256 (SHA-2) are currently considered to be pretty much impossible to crack.

While the mathematical theory allows for the possibility of collisions in checksum hashes, the reality of their application in modern checksum algorithms is substantially different. The sophisticated design of these algorithms significantly reduce the likelihood of such occurrences. This ensures a high level of trust in their effectiveness for generating distinct and reliable representations of data, despite the theoretical potential for identical hashes of different inputs.

Choosing an appropriate checksum algorithm is paramount due to the rapid evolution of computational power as described by Moore's Law, which leads to previously secure algorithms becoming vulnerable as computing capabilities expand.

For instance, MD5 checksums, once deemed secure for storing passwords, are now easily compromised through brute force attacks. This underlines the need for an adaptable approach to checksums, continually updating them to stay ahead of advancements in computational attack strategies. According to L. Courtès [75, Notes on SHA-1, p.16], the SHA-1 algorithm family is now approaching end of life.

To ensure the highest level of security and adaptability to future computational capabilities, it is advisable to use SHA-2 algorithm family such as SHA-384 or SHA-512. These algorithms provide a longer bit length, offering enhanced security and a lower risk of collisions, making them well-suited for securing sensitive data in the face of evolving technological threats.

2.2.4.7. Reproducibility

The concept of reproducibility can be applied in many situations, this thesis will concentrate on a particular application area, thus narrowing its scope. In this thesis, a computation will typically refer to the process of compiling source code into a binary file, except in cases where it is explicitly defined differently.

Reproducibility is a property of a computation. It is the ability to consistently obtain identical results across multiple runs of a computation.

Definition 13: Reproducibility

Reproducibility is a property of a computation satisfying the following condition:

$$\forall c \in C, \forall i \in I, \forall e_1, e_2 \in E, \quad \text{eval}(c, i, e_1) = \text{eval}(c, i, e_2)$$

where
- C is the set of all possible computations
- I is the set of all possible inputs arguments
- E is the set of all possible execution environments (hardware, software, space, time)

Once that condition is met, the computation is considered to be reproducible.

The set I and E, respectically representing the set of all possible inputs and the hardware and software environment including the date and time, are also considered as abstractions. In reality, these sets are complex and intricate entities, that could potentially be composed of many interdependent components. However, for the purpose of this definition, they are treated as atomic.

We could consider expanding the list of arguments to achieve greater specificity, delving deeper into the intricate details that influence reproducibility. However, the objective here is to provide the reader with an initial understanding of reproducibility through a formal definition. This approach is about finding a balance between comprehensive detail and conceptual clarity, thereby offering a foundational glimpse into the formalism that underpins reproducible computational processes without becoming mired in excessive complexity.

The definition of reproducibility (Definition 13) closely matches the definition of pure function (Definition 11) and, inherently, mathematical functions. However, as seen in Table 3, understanding the nuances between theoretical functions and practical computations is essential. Theoretically, mathematical functions are conceptualised as $I \to R$, reflecting the abstract nature of mathematics where the function's result R is purely dependent on its inputs I and external factors are considered non-existent. In the practical world, this theoretical construct is transposed through an evaluation function (Definition 10). For mathematical functions, this environment parameter is known and intentionally left empty, symbolising the deliberate exclusion of external influences and striving to maintain the purity of the theoretical definition. This is in contrast to practical computations in programming, where the environment parameter E is often filled with various real-world parameters and factors, reflecting the nature of computations where outcomes are influenced by the environment variable.

Table 3: Nuances between functions and computations

	Theoretically	**Practically**
Function	$I \to R$	$\text{eval}(F, I, \emptyset) \to R$
Computation	$I \times E \to R$	$\text{eval}(F, I, E) \to R$ Reproducible if and only if $\forall e_1, e_2 \in E \quad \text{eval}(F, I, e_1) = \text{eval}(F, I, e_2)$

This fundamental distinction underscores the challenges of achieving reproducibility and predictability in the practical realm, necessitating robustness and adaptability to manage the variability and complexity of real-world conditions. Together, these definitions provide a comprehensive paradigm for understanding the interplay between the idealised theoretical constructs and their practical applications, emphasising the importance of environmental control in ensuring the computations' reproducibility. The concept of reproducibility, a computational property, underscores the ability to replicate results across different environments within E, serving as a cornerstone for verifying and validating scientific work.

The process of controlling the computational environment E underscores a fundamental challenge in SE: achieving reproducibility through environment standardisation. The environment often encompasses factors such as hardware and software configurations, (CPU, OS, library versions, and runtime conditions), which can significantly impact a function's behaviour and output. The Monte Carlo simulation algorithm (Listing 1), exemplifies this challenge: it may be reproducible at build time but can exhibit variance at run time due to environmental factors.

This singularity highlights the essence of reproducibility: the need to meticulously control or normalise the environment in which computations occur. By ensuring that ideally environment remains constant, we can more closely approximate the behaviour of pure computations in practical software systems. This approach does not merely aim to simplify the computational model but serves as a strategic endeavor to minimise the unpredictability introduced by varying environments.

In conclusion, while the formalism of computations' purity and reproducibility provides the basis of a theoretical framework, the practical application in SE involves the intricate task of environment management. It is through this lens that we understand reproducibility not just as a characteristic of the function itself, but as a holistic property of the entire computational ecosystem, encompassing both the function and its operating environment. This broader view acknowledges that while pure functions offer a paradigm for reproducibility, achieving this in complex, real-world systems often necessitates rigorous control and standardisation of the computational environment which is virtually impossible to deliver.

2.2.5. Software Security

The concept of reproducibility is pivotal in software security for several reasons. Reproducibility ensures that software can be consistently recreated or regenerated from its source code, guaranteeing that the software's behaviour remains unchanged across different builds. This consistency is crucial for verifying the security of software systems. If a software build is reproducible, security experts can confidently assess that the build has not been tampered with or altered to include malicious code. This becomes increasingly important in an era where cybersecurity threats are both sophisticated and prevalent.

In the context of software security, reproducibility also aids in the traceability and verification of software components. It allows for the thorough examination and validation of all parts of the software, ensuring they are exactly as intended and free from vulnerabilities or unauthorised alterations. This traceability is particularly relevant in light of the executive order 14028, *Improving the Nation's Cybersecurity*, issued by Joe Biden [76]. This document underscores the importance of enhancing cybersecurity across federal agencies and emphasises the integrity of the software supply chain.

The european counterpart, the Cyber Resilience Act (CRA) by the European Union reinforces these efforts by setting cybersecurity requirements for software. This act aims to reduce vulnerabilities in software products, enhancing security throughout their lifecycle. Software must come with clear information on their features and instructions for secure installation, operation, and maintenance. This strategy reflects a commitment to producing and using reproducible software.

2.2.5.1. Software Bill Of Materials

The SBOM is an essential element, acting as a detailed inventory of all the components required to build and operate a piece of software, including all applied patches and licensing information in a structured and well-known format.

There are multiple existing formats and standards, the most common ones are:

- Software Package Data Exchange (SPDX): A comprehensive standard maintained by the Linux Foundation, designed to facilitate license compliance, security, and broader software component analysis through a detailed documentation approach, supporting multiple formats like RDF, JSON, and YAML. It caters to a wide range of stakeholders, including software companies, legal teams, and open-source projects, with a particular strength in granular licensing details.
- CycloneDX: A lightweight SBOM standard aimed at enhancing application security and managing software supply chain risks. It emphasises simplicity and efficiency, supporting formats such as XML, JSON, and ProtoBuf, and is particularly tailored towards the identification of software components, their vulnerabilities, and risk assessments, making it a favorite in the application security and DevSecOps communities.

The key differences between the SPDX and CycloneDX formats lie primarily in their focus, structure, and community support. The choice between SPDX and CycloneDX should be guided by an organisation's specific needs, whether the focus is on extensive licensing compliance or streamlined security and risk management within the software supply chain.

2.2.5.2. Supply Chain

A software application is composed of many components, each of which is developed by different teams or organisations. These components are then composed together into a final product, which is the software application itself. This process is known as the *software supply chain* [77].

Contemporary software development leverages the concepts of composability and reusability, preferring the integration and reuse of existing libraries over developing new functionalities from scratch. This methodology enhances productivity and contributes to the creation of more reliable software by allowing each component to concentrate on executing a specific function effectively. Nevertheless, this reliance on external components leads to the accumulation of both direct and indirect dependencies, complicating the software supply chain significantly. The build environments, which encompass all necessary components and their precise versions for software compilation, become intricate and challenging to replicate across different systems and over time. This complexity is often described as *dependency hell*. While Semantic Versioning (Chapter 2, subsection 3.1.4) offers a strategy to mitigate these issues, it alone is insufficient to ensure reproducibility [78, p.11].

To illustrate this concept, the graph in Figure 12 acts as a simplified SBOM for "My App" version `1.2.3`, highlighting its runtime dependencies essential for the application's operation. This visualization selectively excludes the build-time dependencies required for the application's compilation to maintain conciseness. A vulnerability has been identified in `xz` (marked in red), a critical runtime dependency. Consequently, this vulnerability could potentially compromise its dependent components (marked in orange), including the application itself, underscoring the interconnected risk within the software's dependency graph. This scenario, while being a simplified representation, mirrors the recent CVE-2024-3094 [79] in the `xz` project [80], which affected numerous software applications and highlighted the criticality of managing software supply chain risks.

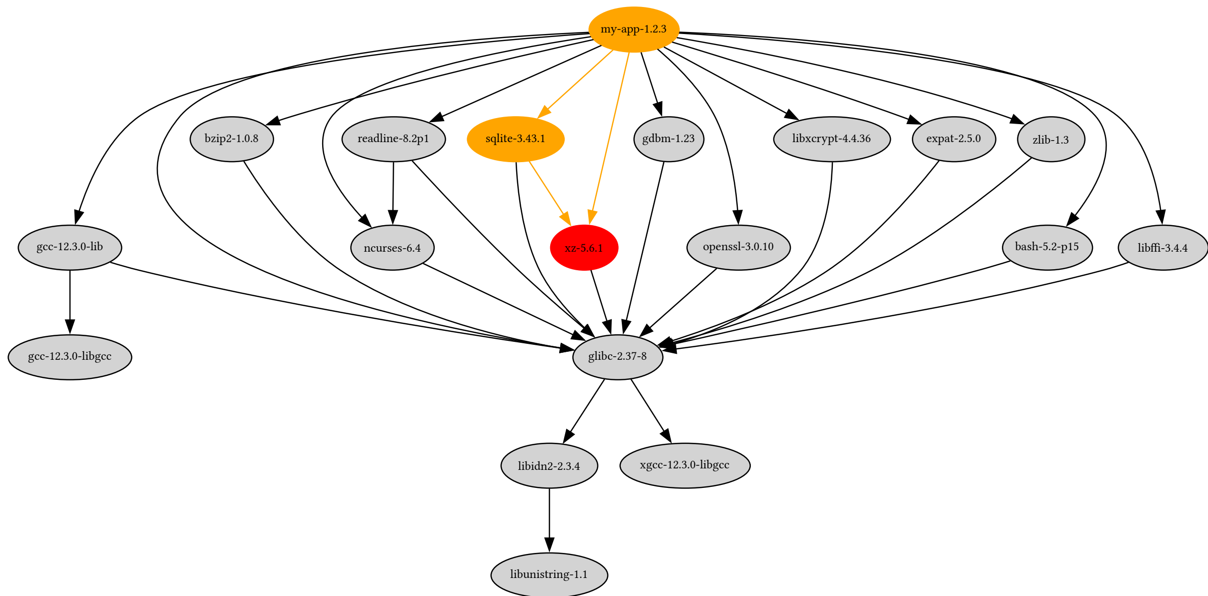


Figure 12: Dependency graph of `my-app` version `1.2.3`, where a flaw has been detected in `xz` dependency

These issues are known as *supply chain attacks*, a type of cyber attack that targets vulnerabilities in the supply chain of software or hardware products, with the aim of compromising the final product by infiltrating its development or distribution process. This can involve tampering with the production of components, the assembly of systems, or the delivery of software updates, thereby infecting end users who trust these sources. One particular aspect of supply chain attacks is that even the original authors of the software may be unaware that their product has been compromised, as the malicious alterations often occur downstream in the supply chain. Although not as frequent as direct attacks on software or systems, supply chain attacks are becoming increasingly common due to their potential for widespread impact. Gartner predicts that by 2025, 45% of organisations worldwide will have experienced attacks on their software supply chains, a three-fold increase from 2021[1] while Cybersecurity Ventures predicts that the global cost of software supply chain attacks to businesses will reach nearly $138 billion by 2031[2]. Notable examples include the *Stuxnet* worm in 2010 [81], the *Heartbleed* bug discovered in 2014 [82], and the *SolarWinds* breach in 2020 [23]. These incidents highlight the exploitation of interconnectedness and inherent trust within the supply chain, making supply chain attacks particularly insidious and effective methods of cyber warfare that can simultaneously affect a large number of users or organisations.

2.2.5.3. Reproducibility And Security

Reproducibility is a fundamental aspect of software security, particularly in the context of the software supply chain. It ensures that software can be reliably and consistently regenerated from its source code, thereby safeguarding against malicious alterations or tampering. This is particularly relevant in the context of supply chain attacks, where the integrity of the software supply chain is compromised, potentially leading to widespread security breaches.

It is paramount to have a clear understanding that having something reproducible doesn't mean that it is secure. It is a necessary condition but not a sufficient one. If a compiler is flawed, it might produce reproducible builds that could also be potentially insecure.

[1] https://www.gartner.com/en/newsroom/press-releases/2022-03-07-gartner-identifies-top-security-and-risk-management-trends-for-2022

[2] https://go.snyk.io/2023-supply-chain-attacks-report.html

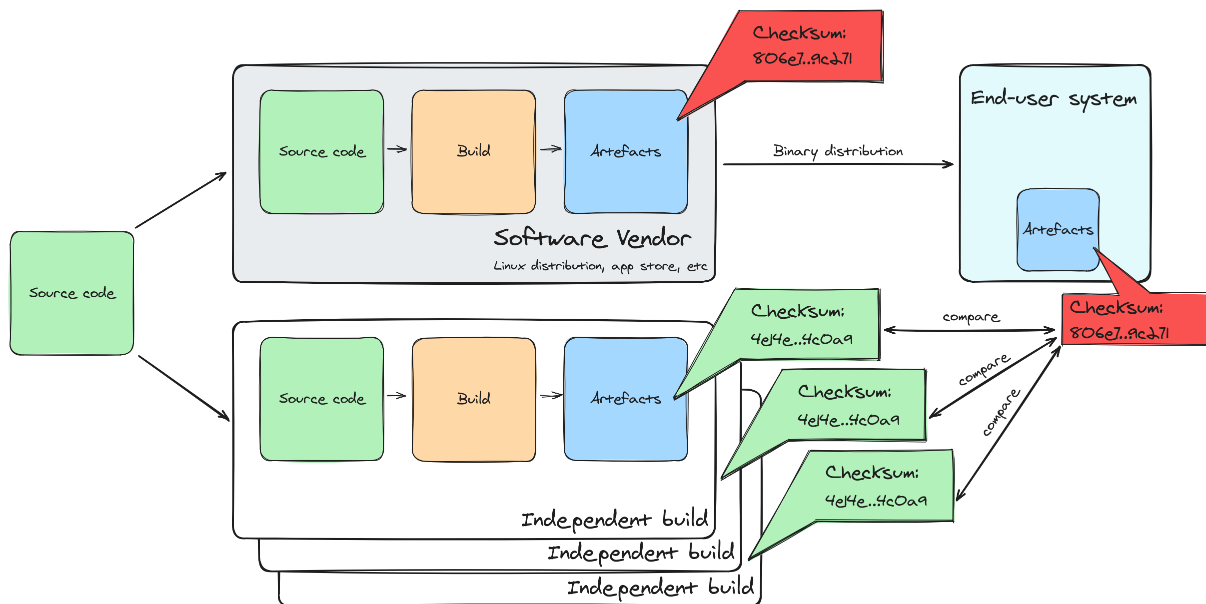


Figure 13: The reproducible builds approach to increasing trust in executables built by untrusted third parties.

In Figure 13 inspired from C. Lamb and S. Zacchiroli [83], end-users should disregard the binary artefact supplied by their software vendor if its checksum (`806e7...9c271`) diverges from those generated by independent third parties (`4e14e...4c0a9`). The security of software is deemed more robust when its reproducibility is confirmed across multiple environments. It is the consensus among these environments that contributes to the perception of security. The premise here is not merely the reproducibility, but the uniformity of this reproducibility across space and time, which strengthens the trust in the software's integrity and security.

As cyber threats evolve, ensuring that software can be reliably and consistently bit-per-bit reproduced from its source code becomes a cornerstone for maintaining security integrity. Reproducibility not only facilitates the verification of software for tampering or malicious alterations but also strengthens trust in software systems amidst the growing complexity of cyber threats. Therefore, integrating reproducibility into software development and distribution processes is a crucial step towards enhancing overall cybersecurity resilience and safeguarding against the ever-increasing sophistication of cyber attacks.

2.2.6. Reproducibility Utopia

Reproducibility in SE is often considered as an utopia. The exact replication of a software poses a significant challenge. Thus, while striving for reproducibility is essential, achieving absolute reproducibility is frequently unattainable in practice.

One of the primary impediments in achieving reproducibility lies in the dependency on hardware architecture. Software compiled for different architectures, such as `x86` and `ARM`, inherently produces disparate binaries [84]. These differences stem from the instruction sets and optimizations that are specific to each platform, leading to divergent outputs despite using identical source code. This variance highlights a significant reproducibility challenge, as achieving bitwise identical results across architectures is **not feasible** as of today.

Compilers (e.g., GCC, Rustc, LaTeX, Typst) also play a role in software development, transforming high-level code into machine-level instructions. However, not all compilers operate deterministically. In this context, non-determinism refers to the phenomenon where compilers produce different outputs given the same input source code across different compilations. Factors contributing to this non-determinism include variations in memory allocation, inclusion of timestamps, and embedding of file paths in the binary output. These variances pose challenges to achieving consistent, reproducible builds.

A compiler is essentially an application that transforms input into output. Tools like GCC are referred to as compilers because they convert high-level code into machine-level instructions. However, the term *compiler* is not limited to programming languages alone. For example, LaTeX is a compiler that transforms a `.tex` file into a `.pdf` file, rustc compiles a `.rs` file into a binary file, and Typst compiles a `.typ` file into a `.pdf` file. Typically, compilers convert human-readable files into machine-readable files.

In Chapter 3, acknowledging the reality that full reproducibility may not be entirely achievable, we will delve deeper into these challenges by exploring the impact of non-deterministic compilers and the strategies to mitigate these challenges using different methods.

2.3. Deterministic Builds And Environments

In this section, we will explore the concept of deterministic builds, and the potential sources of non-determinism in software builds.

The concept of deterministic builds is essential for ensuring reproducibility. A build is termed *deterministic* when it consistently generates identical outputs from a given set of inputs, irrespective of the environment or time of execution. This predictability is central to software reproducibility, yet several sources of non-determinism frequently challenge its realisation. One single non-deterministic component in a build process can render the entire build non-deterministic. Therefore, it is crucial to identify and understand these sources of non-determinism to ensure reproducibility. Many of these sources of non-determinism are related to the environment in which the build occurs. This environment encompasses the hardware, software, and runtime conditions in which the build process is executed. These factors can significantly influence the build process, thereby impacting the stability of the output.

Definition 14: Deterministic build

Let B be a build process defined as a function:

$B : I \times E \to O$

where
- I is the set of all possible input arguments
- E is the set of all possible execution environments (hardware, software, space, time)

then the build B is deterministic if $I \times E$ is deterministic:

$\text{Determinism}(I \times E) \to \text{Determinism}(B)$

where `Determinism` is a function asserting that its argument is deterministic.

According to C. Lamb and S. Zacchiroli [83], a reproducible build environment is essential for achieving deterministic and reproducible builds. It ensures consistency in the software building process by providing a controlled and predictable set of conditions under which the software can be built. J. Malka, S. Zacchiroli, and T. Zimmermann [69, p.1] further elaborate that a build environment is reproducible in both space and time when it is possible to replicate the same build environment on any machine and at any point in the past or future.

When a process exhibits a lack of reproducibility over time, it indicates a fundamental instability within the process. While it would be technically feasible to replicate the same output in a different environment, within the same architecture, achieving exact temporal replication of the build process is practically impossible. This temporal variability serves as a critical indicator of potential difficulties in ensuring reproducibility across diverse environments or machines.

2.3.1. Sources Of Non-Determinism

In this section we will explore the sources of non-determinism in software builds and usage. The list is not exhaustive, it just includes the most common sources of non-determinism. The list is created from C. Lamb and S. Zacchiroli [83]'s paper and information of the Reproducible Builds [46] project, a website aiming at improving reproducible builds in software development.

2.3.1.1. Randomness

Using random data in a computation is a common source of non-determinism and must be avoided. When random data is required, the solution is to use a predetermined value acting as a seed to the pseudo-random number generator. Using a predetermined value as a seed ensures that the same random data is generated each time the computation is executed, thereby guaranteeing reproducibility.

Hardcoding the seed in the source code would be nonsensical because it wouldn't be random anymore, the seed should be passed as a parameter to the computation. This parameter can be passed as a command-line argument, an environment variable, or a configuration file, leaving the responsibility to the user to provide a seed.

2.3.1.2. Build Paths

Build paths are paths used by the source code to locate files and resources. Sometimes, it can happen that absolute paths are used in the source code, which means that the build will only be reproducible on the same machine where it was built.

To avoid this, relative paths should be used instead of absolute paths and sometimes post-processing is required to remove the build path or to normalize it with a predefined value.

2.3.1.3. Volatile Inputs

Volatile inputs are inputs that can change over time. For example, the current date and time are volatile inputs, network streams as well. Dealing with date and time will be done in Chapter 2, subsection 3.1.7. For network streams, the solution is to never rely on remote data while building. Instead, the data should be downloaded beforehand and stored locally.

This is a common issue in the context of software compilation, where the build process might download dependencies from the internet during the build.

2.3.1.4. Package Managers

Package managers are tools that automate the process of installing, upgrading, configuring, and removing packages, typically from a central repository or package registry. They are widely used in software development to manage dependencies and facilitate the build process. For example, `Cargo` for Rust, `Composer` for PHP, `NPM` for NodeJS, `Dune` for OCaml. They are also used to manage software at the operating system level like: `apt` in Debian based distributions, `pacman` in Arch Linux, `dnf` in Fedora, `brew` in MacOS, `chocolatey` in Windows.

Package managers can inadvertently introduce non-determinism by automatically downloading or updating dependencies to their latest versions. This process can lead to inconsistencies, particularly when a newer version of a package includes changes that are incompatible with the project's codebase. To mitigate this, the Semantic Versioning (SemVer) scheme is widely adopted, offering a structured version naming convention that aids dependency management. However, while packages may declare SemVer compliance, adherence levels vary, with some strictly following SemVer principles and others adopting them more leniently [78, p.5]. Notably, there has been a trend towards increasing adoption and stricter adherence to SemVer principles by package managers over time [78, p.13]. It provides a structured version naming convention designed to convey the nature of changes between releases, thereby aiding in the management of dependencies with a syntax that succinctly specifies version constraints. While this mechanism greatly facilitates dependency resolution by leveraging a minimalistic syntax, it inherently permits variability over time, potentially compromising reproducibility.

```json
{
    "name": "awesome/php-library",
    "require": {
        "foo/http": "^1",
        "foo/bar": "1.2.3"
    }
}
```

Listing 4: A `composer.json` file, used by the PHP package manager, Composer

In Listing 4, the dependency `foo/http` is specified with version `^1`, where the caret symbol (`^`) indicates that Composer should install the latest minor version within the major version `1`. In contrast, the dependency `foo/bar` is locked to version `1.2.3`, signalling that Composer must install that specific version, regardless

of newer releases. This distinction underscores the importance of using package managers judiciously to achieve determinism. For Composer, determinism is further ensured by including a `composer.lock` file in the project, which explicitly pins each dependency to a particular version, thus facilitating reproducibility. The decision to require this file varies by project and is not in the scope this master thesis.

Ensuring reproducibility in the context of package managers is particularly challenging due to the amount of different ecosystems and the lack of standardisation. For example, in the Python realm[3], there have been and there are still multiple package manager ecosystems: `distutils`, `setuptools`, `pip`, `pypi`, `venv`, `conda`, `anaconda`, `poetry`, `hatch`, `rye`. Each of these has its own configuration file format, which can be used to specify the version of each dependency. However, there is no standardisation which makes it difficult to ensure reproducibility. The same issue applies to operating system's package managers. For example, in Debian based distributions, there are multiple package managers: `apt`, `aptitude`, `dpkg`.

The solution would be to use a universal package manager that would work for all Linux distributions and programming languages. This is what tools like `AppImage`, `snap` and `flatpak` are trying to solve, only at the level of the operating system. These tools are partially fixing the issue by just being available only for installing package at the operating system level.

These tools, while being a step in the right direction, are also coming with their own set of issues, like the lack of standardisation between them, the lack of adoption and the lack of support from major distributions.

There are also package managers like `Nix` and `Guix` that are trying to solve the issue by being universal. They provide a way to build and install packages in a sandboxed environment, which means that packages are isolated from the rest of the system at build time. This is a great way to ensure reproducibility, we will discover them in Chapter 4

2.3.1.5. Version Information

Version information like commit identifiers can be used to precisely identify the source code used to build a program.

```
$ typst --version
typst 0.10.0 (f2433bd1)
```

Listing 5: Example of program including a commit ID

As illustrated in Listing 5, incorporating specific version information, such as a commit ID, helps reproduce a build by facilitating the retrieval of an identical source code version. Nevertheless, the efficacy of commit IDs as reproducibility anchors remains debatable. These identifiers may frequently be unavailable at the time of build. It is essential to recognize that `git`, a distributed version control system designed to handle everything from small to very large projects with speed and efficiency, metadata, including commit IDs, is not an intrinsic element of the source code. Instead, it is part of the version control system in use. `git` allows multiple developers to work together on the same project simultaneously, providing a robust system for tracking changes, version history, and collaboration. However, the potential for easy substitution of one version control system for another renders reliance on such ephemeral metadata a precarious foundation for software reproducibility.

In scenarios where a version number is necessary, it can be derived from a dedicated file, such as a changelog or eventually provided through an environment variable. This approach decouples the versioning process from the underlying version control system, potentially offering a more stable and reliable method for software version identification.

2.3.1.6. File Order

It is important to ensure that processing multiple files in a stable order remains stable.

Listing files relies on the low-level POSIX call `readdir`, which itself is dependent on the filesystem in use and therefore doesn't guarantee any consistent ordering.

[3]https://linuxfr.org/news/l-installation-et-la-distribution-de-paquets-python-1-4

According to S. Loosemore, R. Stallman, R. McGrath, A. Oram, and U. Drepper [85, p.415]: The order in which files appear in a directory tends to be fairly random. A more useful program would sort the entries before printing them.

In M. Kerrisk [86, p.354]: The filenames returned by `readdir()` are not in sorted order, but rather in the order in which they happen to occur in the directory. This depends on the order in which the file system adds files to the directory and how it fills gaps in the directory list after files are removed.

There are numerous situations where relying on an existing list of files can result in non-determinism. For instance, when generating an archive from the contents of a directory, many file systems do not provide consistent ordering when listing files within that directory. Consequently, the arrangement of files in the archive may differ between builds, causing unpredictable archives. Although these archives might contain identical content, they could have been compressed with varying file orders.

To address this, one could enforce a stable order by explicitly sorting the inputs before processing them. This can be done by sorting the list of files in the directory based on a specific criterion, such as their names or modification timestamps.

```
1 $ tar --sort=name -cf archive.tar /tmp
```

Listing 6: Use of `--sort=name` flag to ensure a stable order of files in an archive

2.3.1.7. Timestamps

Timestamps are among the biggest sources of non-determinism in software builds, as they can lead to differences due to changing times between builds. Since reproducibility checks the content of the output and its metadata, building multiple times some source code will create output artefacts with possibly the same content but with different metadata, like file timestamps, making them irreproducible.

Often, timestamps are used to approximate which version of the source were built. Since file timestamps are volatile, the source code needs to be tracked more accurately than just a timestamp. Just like for version information, the solution would be to extract the date from a dedicated file like a changelog, or a specific commit [87].

To circumvent this issue, `SOURCE_DATE_EPOCH` is a specific environment variable convention for pinning timestamps to a specific value that has been introduced by the `reproducible-builds.org` community and it is now widely used by many compilers and build tools.

Another option is to use `libfaketime`, a library that intercepts system function calls retrieving the current time of day and replies with a predefined date and time instead.

When none of these options are viable, using a tool like `strip-nondeterminism` [88] is a temporary workaround for stripping non-deterministic information such as timestamps and filesystem ordering from various file and archive formats.

2.3.1.8. Locale Environment Variables

```
1 $ date -d@2147483647
2 Tue Jan 19 04:14:07 AM CET 2038
3 $ date -d@2147483647 -u
4 Tue Jan 19 03:14:07 AM UTC 2038
5 $ LC_ALL=C date -d@2147483647 -u
6 Tue Jan 19 03:14:07 UTC 2038
```

Listing 7: Use `LC_ALL` and `-u` flags to configure the date format

`LC_ALL` is a locale environment variable that can modify various aspects of an application's behaviour. It can change the date format, string collation order, and character encoding. Although each parameter can be set individually, `LC_ALL` enables you to configure them all simultaneously and override any other locale environment variables.

In Listing 7, we methodically incorporate various flags, such as `-u`, and the `LC_ALL` environment variable to the `date` command. This approach ensures that the output we receive is predictable and consistent, regardless of the underlying system configuration.

2.3.2. Comparing Builds

In the quest for software reproducibility, identifying and understanding the differences between two builds of the same software becomes paramount, especially when those builds are not identical. This section introduces a tool designed specifically for this purpose.

Developed under the umbrella of the Reproducible Builds [46] effort, `diffoscope` [89] is a comprehensive, open-source tool that excels in comparing files and directories. Its unique capability to recursively unpack archives of various types and transform binary formats into a human-readable form makes it an indispensable tool for software comparison. It seeks to simplify the process of identifying discrepancies between software builds. This functionality is crucial for developers and researchers striving to pinpoint and resolve the causes of non-reproducibility. An online version of the tool is also available[4].

To demonstrate the effectiveness of `diffoscope` in identifying differences between non-reproducible builds, Listing 8 considers the hypothetical example of a simple program that outputs the current date and time. Due to its nature, compiling this program twice, even with the same source code, will inherently produce two different builds.

First, we compile the sourcecode twice, creating `build1` and `build2`:

```
$ gcc -o build1 datetime.c
$ gcc -o build2 datetime.c
$ sha256sum build*
5d0b1ae966c719862971bd51b4c035a478863a409caa605b2c529d45f2ac137d  build1
7cbfd989b49c7336dc495a055db223253f0c1d6dc232b66b0fe0f7b6103c274c  build2
```

Listing 8: Compilation of non-reproducible programs and the use of their checksums for comparison

Then, we use `diffoscope` to compare these builds:

```
$ diffoscope build1 build2
```

The tool will generate a detailed report (Figure 14) highlighting the differences between `build1` and `build2`. In this hypothetical example, differences might include timestamps or other build-specific metadata embedded within the binary.

[4] https://try.diffoscope.org/

```
build1 vs.                                                          1010 B
build2

strings --all --bytes=8 {}                                           518 B
Offset 1, 16 lines modified              Offset 1, 16 lines modified
1   /lib64/ld-linux-x86-64.so.2           1   /lib64/ld-linux-x86-64.so.2
2   __libc_start_main                     2   __libc_start_main
3   libc.so.6                             3   libc.so.6
4   GLIBC_2.2.5                           4   GLIBC_2.2.5
5   GLIBC_2.34                            5   GLIBC_2.34
6   /lib:/usr/lib                         6   /lib:/usr/lib
7   __gmon_start__                        7   __gmon_start__
8   18:23:34
9   Feb 1 2023                            8   15:41:32
                                          9   Mar 2 2024
10  Built the %s at %s.                   10  Built the %s at %s.
11  GCC: (GNU) 13.2.0                     11  GCC: (GNU) 13.2.0
12  __abi_tag                             12  __abi_tag
13  crtbegin.o                            13  crtbegin.o
14  deregister_tm_clones                  14  deregister_tm_clones
15  __do_global_dtors_aux                 15  __do_global_dtors_aux
16  completed.0                           16  completed.0

readelf --wide --decompress --hex-dump=.rodata {}                    483 B
Offset 1, 6 lines modified               Offset 1, 6 lines modified

1   Hex dump of section '.rodata':
2   0x00402000 01000200 31383a32 333a3334 00466562 ....18:23:34.Feb
3   0x00402010 20203120 32303233 00427569 6c742074   1 2023.Built t
4   0x00402020 68652025 73206174 2025732e 0a00       he %s at %s...

1   Hex dump of section '.rodata':
2   0x00402000 01000200 31353a34 313a3332 004d6172 ....15:41:32.Mar
3   0x00402010 20203220 32303234 00427569 6c742074   2 2024.Built t
4   0x00402020 68652025 73206174 2025732e 0a00       he %s at %s...
```

Figure 14: A `diffoscope` report using HTML format

2.3.3. Fixing Builds

In this subsection, we delve into strategies for addressing non-reproducible builds, acknowledging the vast array of potential causes and the impossibility of covering every solution comprehensively.

Previously in Listing 8, we encountered an issue, where compiling the sourcecode (Listing 2) twice resulted in different binaries. Using `diffoscope`, we identified, as shown in Figure 14, the source of variability as date and time strings embedded within the binaries.

A solution has been proposed in Chapter 2, subsection 3.1.7, we can leverage the `SOURCE_DATE_EPOCH` environment variable to address this specific challenge in achieving reproducible builds. This approach standardises the date and time used during the build process, ensuring consistency across compilations and thus contributing to reproducibility.

```
1 $ export SOURCE_DATE_EPOCH=1709373544
2 $ gcc -o build1 datetime.c
3 $ gcc -o build2 datetime.c
4 $ sha256sum build*
5 98f0419783bb3b06b45eaf5cc0efeef9408c4dad1e9eec9eb153dc7a6cc6962f  build1
6 98f0419783bb3b06b45eaf5cc0efeef9408c4dad1e9eec9eb153dc7a6cc6962f  build2
```

Listing 10: Fix builds using an environment variable

2.4. Conclusion

This chapter embarked upon a detailed journey through the landscape of reproducibility, focusing particularly on its pivotal role within the realms of science and, more specifically, CS and SE. Through rigorous analysis, we unveiled the multifaceted nature of reproducibility.

We dissected the concept of reproducibility, from its foundational elements in science to its intricate implications in computer science, delineating the essential terminology that frames our discussion: computations, pure and impure functions, inputs, outputs, and the environmental variables that intertwine to influence reproducibility. The exploration into deterministic builds and the sources of non-determinism not only highlights the inherent challenges but also sets the stage for the subsequent focus on the tools and methodologies designed to tame these complexities.

As we pivot toward the next chapter, our narrative will transition from the theoretical underpinnings to the practical arsenal at our disposal for enhancing reproducibility in SE. While the groundwork laid in this chapter paves the way for an in-depth exploration, it is important to acknowledge the vast landscape of tools and methodologies available in this domain. Given the scope of this thesis, we will focus on four evaluation methods using three key tools, with the understanding that this selection is not exhaustive but rather representative of the broader ecosystem. Through the lens of real-world applications and case studies, we will explore how these chosen tools are used to mitigate the challenges identified herein and to foster an ecosystem where reproducible research and development are not merely aspirational goals but operational norms.

In fine, this chapter serve as both a foundation and a bridge. It offers a comprehensive understanding of reproducibility that is critical for appreciating the significance of the solutions and methodologies discussed in the chapters that follow. It is within this framework that we continue our quest to demystify reproducibility, moving from conceptuals clarity to practical application, with the ultimate aim of enhancing the reliability, security, and transparency of SE practices.

This page is intentionally left blank.

Chapter III

Software evaluation

Any sufficiently advanced technology is indistinguishable from magic.

— A. C. Clarke [90]

This chapter explores the pivotal role of tooling in achieving reproducibility within SE, highlighting the importance of environment consistency, dependency management, and process isolation.

Reproducibility in SE is not merely a desirable attribute but a cornerstone of trustworthy, reliable, and verifiable software development practices. As software systems grow increasingly complex and integral to every facet of the modern world, from critical infrastructure to personal devices, the stakes for ensuring their reproducibility have never been higher. This chapter introduces and examines four distinct methods for building software, each with its unique approach:

- Bare compilation

  It is the most rudimentary method, depends on the operating system's compilers and libraries for software construction.

- Compilation with Docker

  Using containerization technology, encapsulates not just the software and its dependencies but also the entire runtime environment.

- Compilation with Nix

  Nix uses a unique store for packages built in isolation, each with a unique identifier that includes dependencies, preventing conflicts and ensuring reproducible environments.

- Compilation with Guix

  Inspired by Nix, Guix offers a transactional package management system that isolates dependencies to ensure consistent and reproducible software environments through specific version-linked profiles.

The four evaluation methods chosen for detailed evaluation in the context of reproducibility represent a wide range of approaches to managing software build environments, each addressing different aspects of reproducibility. Bare compilation was selected to provide a baseline, demonstrating the fundamental challenges encountered without the aid of advanced tooling, such as environmental inconsistencies and dependency conflicts. This method serves as a contrast to the more sophisticated techniques that follow. Docker is included for its widespread adoption and popularity, as well as its approach to encapsulating the runtime environment, which significantly mitigates issues arising from system variability. Guix and Nix are examined due to their unique approach to dependency management and environment isolation, employing a package management approach that is based on the functional paradigm (Definition 15) to ensure exact reproducibility of environments across different systems. The chapter aims to cover a spectrum from the most basic to the most advanced strategies.

Definition 15: Functional package management

From L. Courtès and R. Wurmus [91], functional package management is a discipline that transcribes the functional programming paradigm to software deployment: build and installation processes are viewed as pure functions (Definition 11) in the mathematical sense whose result depends exclusively on the inputs (Definition 9), and their result is a value that is, an immutable directory.

This chapter aims to provide readers with an understanding of how these contribute to the broader goal of reproducible SE. Through a detailed exploration of each approach, readers will gain insight into the strengths, weaknesses, and applicability of Bare compilation, Docker, Guix and Nix in various software development scenarios.

3.1. Methodology

Our primary objective is to assess the reproducibility of a software build using four different methods: Bare compilation, Docker, Guix, and Nix. By compiling a C program (Listing 2) with each tool, we can evaluate reproducibility both over space and time (Definition 5).

The study uses a quantitative research design, focusing on the comparison of binary files generated from compiling identical source code with different methods, on the same environment. This approach allows for an empirical assessment of reproducibility challenges inherent to each compilation tool and environment.

3.1.1. Evaluation Criteria

We will consider three primary criteria.

Firstly, **reproducibility in time** assesses whether the outputs of builds are identical across repeated compilations in the same environment. This criterion involves compiling the same source code twice with a few seconds of interval between compilations. By comparing the outputs of these compilations, we can determine if the build process produces consistent results over time.

Secondly, **reproducibility in space** focuses on the consistency of build outputs across different environments. To evaluate this, the same source code is compiled in various environments. This process helps to ensure that the software build process is not dependent on specific environmental factors and can produce identical outputs regardless of where it is compiled.

Lastly, the **reproducibility of the build environment** evaluates the stability and consistency of the environment itself, including the dependencies required for building the output. This criterion ensures that the environment, which encompasses all necessary tools and libraries, remains stable and consistent across different instances and setups.

3.1.2. Tools And Technologies

The evaluation of reproducibility tools in this study encompasses several approaches to software compilation and package management, each with its unique methodology.

In Chapter 3, section 2, the bare compilation method involves direct compilation on the host system without the use of containerization or package management tools.This approach relies on the default tools and libraries installed in the operating system, providing a straightforward but less controlled environment for building software. This method is assessed to understand the baseline reproducibility and potential variability introduced by the host system's native environment.

In Chapter 3, section 3, Docker is used to provide a containerized environment for software compilation. Using Docker containers ensures that the build process occurs in a consistent and isolated environment, independent of the host system's configuration. This method helps in evaluating how containerization can enhance reproducibility by encapsulating all necessary dependencies and tools within a controlled and replicable environment.

In Chapter 3, section 4, the Guix package ecosystem is employed to manage the software build process. Guix focuses on providing a reproducible and declarative approach to package management, ensuring that the build environment and dependencies are precisely defined and versioned. This approach is examined for its ability to maintain consistency and reproducibility across different systems and environments by leveraging Guix's robust package management features.

In Chapter 3, section 5, the Nix package ecosystem is used to manage and build software. Similar to Guix, Nix offers a declarative and reproducible package management system, allowing for precise control over the build environment and dependencies. The evaluation of Nix focuses on its capability to provide a reproducible build environment that can be consistently replicated across various systems, enhancing the reliability and stability of the software development process.

3.1.3. Scenarios

Our examples and builds focus on custom-made scenarios to highlight the differences in reproducibility across the four tools. There are multiple scenarios being evaluated:

In the first scenario, using Chapter 3, section 2, a C program is built using the host default C compiler. The second scenario involves Chapter 3, section 3, where a C program is built in a Docker container utilizing the C compiler. The third scenario, with Chapter 3, section 4, involves building a C program using Guix. Finally, there are two scenarios for Chapter 3, section 5: one involves building a C program using Nix legacy (not flake), and the other uses Nix flake to build the same program.

3.1.4. Compilation And Execution

A trivial C program (Listing 2) has been chosen for its straightforwardness, allowing the focus to remain on the build process and output rather than software complexity.

Each method will compile the same C program (Listing 2) twice. Detailed steps for compilation and execution within each environment will be documented, ensuring transparency and reproducibility of the

process itself by the readers. Each compilation's resulting output will be executed to verify functionality, although the correctness of the execution's output will not be evaluated.

3.1.5. Environment Setup

To ensure the robustness and universality of our reproducibility assessment, all test scenarios described in this chapter are executed through GitHub Actions [3]. GitHub Actions is an automation platform that enables CI/CD, allowing builds to be performed, tested, and deployed across various machines and architectures directly from GitHub repositories [92].

Our testing environments supports three distinct architectures:

- `x86_64-linux`: This represents the widely used Linux operating systems on Intel and AMD processors. To ensure a thorough evaluation, two instances, each running the different versions of Ubuntu (`20.04` and `22.04`), are employed.
- `x86_64-darwin`: Dedicated to supporting macOS on Intel processors.
- `aarch64-darwin`: Addressing the latest generation of macOS powered by the ARM-based Apple Silicon processors.

This selection encompasses both `x86` and `ARM` architectures, as well as Linux and MacOS operating systems, providing a comprehensive view of reproducibility across the most commonly used development platforms in SE. The choice of these architectures ensures the results are relevant to a broad spectrum of development environments and application targets.

Each of our scenarios is streamlined through the use of a `Makefile`. A `Makefile` as seen in Listing 11 is a text file that contains a set of directives used by the GNU `make` [93] utility to automate the build process of software projects. These directives contain specific shell commands.

```
clean:
    # This `clean` step is a dependency of the build step
    # It will remove the files and artefacts generated by the `build` step
    rm -rf test.txt

build: clean
    # This `build` step will compile the source code and generate the
    # output artefacts.
    # It has a dependency on the `clean` step
    echo "hello world" > test.txt

check:
    # This `check` step will print the checksum of the output artefacts
    sha256sum test.txt

run:
    # This `run` step will execute the output artefacts and print the result
    # In case of a non-executable output, it will print the content of the # file or its checksum.
    cat test.txt
```

Listing 11: An example of `Makefile` used in a scenario.

Each scenario's `Makefile` essentially contain four essential steps:

- `clean`: Removes the build artefact of a build process, if any.
- `build`: Executes a build process, generating an output artefact.
- `check`: Prints the checksum of the build artefact.
- `run`: Execute the artefact

Incorporating these `Makefile` steps into our GitHub Actions workflows not only automates the execution of each scenario, ensuring consistency and repeatability in our testing process, but also empowers the reader with the ability to locally reproduce the steps outlined in this document in full transparency. This approach facilitates and encourages the direct replication of methods and scenarios, aligning with best practices in SE for reproducibility, but also extends these principles to broader scientific research practices.

3.1.6. Output Comparison

To compare the results, we will compare the checksums of the resulting outputs. We exclusively use the `nix hash path` command provided by the Nix package manager to compute the hash of a path.

The `nix hash path` command is provided by Nix, a tool we will explore in this chapter. Nix provides this command as part of its suite, but it can be applied anywhere, not just to files within the Nix ecosystem. This command distinguishes itself by its capacity to hash directories in addition to files. An alternative to this approach could have been the use of a Software Heritage Identifier (SWHID) [15].

The `nix` command is available on systems with Nix installed. The difference with a traditional `sha256sum` is that the former computes the hash of the path, which includes the content and the metadata while the latter computes the hash of the content only. Another advantage of using that command is its ability to create a hash prefixed by the algorithm used, similar to Subresource Integrity (SRI) [14] hashes.

3.1.7. Expected Outcomes

At the opposite of the previous more theoretical chapters, this practical chapter aims to empirically compare the differences in reproducibility achievable with Bare compilation, Docker, Guix, and Nix. Insights into the challenges and benefits of each method will inform best practices in SE for achieving reproducible builds.

3.2. Evaluation 1 - Bare compilation

This method is the most rudimentary approach to software compilation, relying on the host system's installed compilers and libraries to build software. This build method correspond to Scenario 1, with the corresponding `Makefile` in Listing 12, that can be executed on any system, with the commands: `make build` to compile, `make check` to print the checksum, `make run` to run the compiled binary. As explained in Chapter 3, section 1.4, we notice that the steps are executed twice and in Terminal session 4, the steps to build, check and run the build are detailed.

```
1  clean:
2    rm -rf datetime
3
4  build: clean
5    gcc src/datetime.c -o datetime
6
7  check:
8    nix hash path datetime
9
10 run:
11   ./datetime
```

Listing 12: `Makefile` of Scenario 1

```
1  $ cd lib/scenario-1
2  $ make -s build
3  $ make -s check
4  sha256-fnmArVyFozZkU/T0hrABdyq5M1kmAlUXSWvYodJm9bw=
5  $ make -s run
6  Built the Mar 15 2024 at 14:51:27.
7  $ make -s build
8  $ make -s check
9  sha256-aFgPLDbv2BY+rCKDzsE+0fr9ylGr2R4fbWMhr4lfEQo=
10 $ make -s run
11 Built the Mar 15 2024 at 14:52:18.
```

Terminal session 4: Terminal log of the steps to build, check and run Scenario 1

At lines 4 and 9 of Terminal session 4, we notice that the `make check` step prints two different checksums, indicating that the output of the two builds is different at each run. As a result, this build is not reproducible. This discrepancy in the output is likely caused by the dynamic replacement of the `__DATE__` and `__TIME__` macros in the source code, which are replaced with the current date and time at the moment of compilation.

3.2.1. Reproducibility In Time

This method involves directly compiling source code on a system with only the essential compilers and libraries available on the host. This method's primary advantage lies in its simplicity and direct control over the build process, allowing for a clear understanding of dependencies and compilation steps. However, it lacks isolation from the system environment, leading to potential *it works on my machine* issues due to

variations in system configurations. Additionally, the lack of encapsulation and dependency management can lead to difficulties in achieving consistent and reproducible builds across different environments. This method is therefore classified as non-reproducible in time.

3.2.2. Reproducibility In Space

This method is not reproducible in time, therefore we will consider it as not reproducible in space either. Technically it would be possible to reproduce the same output on another environment, but it would be practically impossible to run the build at exactly the same time. This method is therefore classified as non-reproducible in space.

3.2.3. Reproducibility Of The Build Environment

The virtual machines used on Github Actions are versioned. However, the software installed on the images are not. From one build to another, we can have a different version of `gcc` or any other software installed on the image. Therefore, we have absolutely no control over the build environment and it is very complicated to reproduce the same environment on another machine. Therefore, reproducibility of the build environment is not guaranteed.

3.3. Evaluation 2 - Docker

Docker [94] has revolutionised software deployment by encapsulating applications in containers, ensuring they run consistently across any environment. Unlike traditional virtual machines, Docker containers are lightweight, share the host's kernel, and bundle applications with their dependencies, promoting the principle of *"build once, run anywhere"*. This approach streamlines development, testing, and production workflows, significantly reducing compatibility issues and, to some extent, simplifying scalability.

Central to Docker's appeal is its contribution to the DevOps movement, fostering better collaboration between development and operations teams by eliminating the *"it works on my machine"* problem. Docker's ecosystem, including the Docker Hub [70], offers a vast repository of container images, facilitating reuse and collaboration across the software community.

Docker uses the Open Container Initiative (OCI) standard for its container images, ensuring interoperability across different containerization technologies, including Podman [95] and Kubernetes [96]. The OCI specification outlines a format for container images and a runtime environment, aiming to create a standard that supports portability and consistency across various platforms and cloud environments.

Due to its popularity, Docker is a key player in modern software development, enabling efficient, consistent, and scalable applications through containerization, supporting agile and DevOps practices, and accelerating the transition from development to production.

```
FROM alpine@sha256:c5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4f27761f8e1ad6b as build-env
RUN apk add --no-cache build-base
WORKDIR /app
COPY . .
RUN gcc datetime.c -o datetime

FROM alpine@sha256:c5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4f27761f8e1ad6b
COPY --from=build-env /app/datetime /app/datetime
WORKDIR /app
CMD ["/app/datetime"]
```

Listing 13: From Scenario 2, the `dockerfile` used by Docker

This method involves creating an OCI image, compiling Listing 2, through a `Dockerfile` and setting the compilation result as default command as shown in Listing 13. This ensures that each time the image is executed, the compiled executable runs within the container. However, instead of printing only the checksum of the resulting binary, the `check` step also outputs the checksum of the image.

```
$ cd lib/scenario-2
$ make -s build
Loaded image: datetime
$ make -s check
sha256-zV0YiwaUrvtHP8Xyn3wfQqkHkeRJonEI6c7BZVnEKQo=
sha256-VKMisMOGn3qq3/TeGmk2NLWhU4Aez3FigY1TWiZvBOQ=
$ make -s run
Built the Mar 26 2024 at 13:36:07.
$ make -s build
Loaded image: datetime
$ make -s check
sha256-pY/buhRq1FZzVoMsImuWpqS0NONp7VRpinODlDIm1To=
sha256-e1VCDdKYA2WMW9SKBXD/BqzDBSbCdXejLejEUx1XZi4=
$ make -s run
Built the Mar 26 2024 at 13:37:30.
```

Terminal session 5: Terminal log of the steps to build, check and run Scenario 2

3.3.1. Reproducibility In Time

In Terminal session 5, it is observed on lines 5 and 12 that building the image twice and extracting the resulting binary produces different checksums. Additionally, on lines 6 and 13, it is evident that the checksums of the images are inevitably different. Consequently, this method is classified as non-reproducible over time.

3.3.2. Reproducibility In Space

This scenario was executed on various machines and architectures, resulting in different binaries and images. Therefore, this method is classified as non-reproducible in space as well.

3.3.3. Reproducibility Of The Build Environment

The reproducibility of build environments in Docker images, while generally reliable in the short term, can face challenges over time. Docker images are built on layers, often starting from base images provided by specific vendors. These base images can receive updates that alter their contents, meaning a `Dockerfile` that successfully built an image at one time might not produce an identical image later due to changes in its base layers. Additionally, not pinning specific versions of base images and external dependencies in the `Dockerfile` can lead to inconsistencies, making the exact reproduction of a Docker environment challenging if not managed carefully. Therefore, while Docker simplifies the consistency of deployment environments, ensuring long-term exact reproducibility requires careful management of image sources and dependencies.

Docker is intrinsically designed to facilitate reproducible builds, with the capability to generate identical containers across multiple executions. However, the challenge to reproducibility arises not from Docker's fundamental features but from the use of specific base images within Docker containers. A significant illustration of this problem is shown in Terminal session 5, where rebuilding the image results in different containers even though the base image version has been pinned to a specific commit at lines 1 and 7.

"Pinning" refers to the practice of specifying exact versions of software, base images, or dependencies to use when building a Docker container. This practice helps ensure that the build environment remains consistent and predictable over time, despite updates or changes to those dependencies. Pinning is crucial for maintaining consistency as it prevents the build environment from changing unexpectedly due to updates in dependencies. It also enhances reproducibility, allowing developers to recreate the same environment at a later date, which is vital for debugging and development. Moreover, it enhances reliability by reducing the likelihood of encountering unexpected issues or conflicts caused by differing versions of dependencies.

For example, specifying `FROM alpine:3.19.1` in a `Dockerfile` instead of `FROM alpine` ensures that the Alpine 3.19.1 version is always used, providing stability. This mechanism applies similarly across different programming language ecosystems. However, it is important to note that version tags, like `3.19.1`, can be replaced or updated by the maintainers, potentially altering the contents associated with a *pinned* version.

To overcome this, the use of digests or checksums (Chapter 2, subsection 2.4.6) can anchor images to a specific snapshot, offering a stronger guarantee of immutability. For instance, specifying `FROM alpine@sha256:c5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4f27761f8e1ad6b` as shown in Listing 13 ensures that exactly the same image is used consistently, regardless of any updates.

Docker's containerization technology offers a way to create consistent software environments across various systems by encapsulating both the software and its dependencies within containers. This encapsulation aids in ensuring a uniform deployment process. However, the approach's reliance on base images and the package managers they use brings forth challenges in maintaining reproducibility. This is primarily because base images might not be strictly version-controlled, and the package managers used within these images can result in the installation of varying dependency versions over time.

For example, traditional package managers like `apt` (used in Debian-based OSes) or `yum` (used in RedHat-based OSes) do not inherently guarantee the installation of the exact same version of a software package across space and time. Typically, this variability stems from updates in the package repositories, where an `apt-get install` command might fetch a newer version of a library than was originally used. Such updates could potentially introduce unexpected behaviour or incompatibilities.

Docker and similar containerization technologies act as sophisticated assemblers, piecing together the diverse components required to create a container. This process, while streamlined and efficient, is not immune to the introduction of variability at any stage of the assembly line. Whether due to updates in base images, fluctuations in package versions, or differences in underlying infrastructure, these variables can compromise the reproducibility of the resulting container (Definition 14). Recognising this, it becomes crucial for developers and researchers to approach container creation with a keen awareness of these potential pitfalls. By meticulously managing base images, employing reliable package managers, and adhering to best practices in `Dockerfile` construction, one can mitigate the risks of variability and move closer to achieving true reproducibility in containerised environments.

3.4. Evaluation 3 - Guix

Guix [97] is an advanced package manager, designed to provide reproducible, user-controlled, and transparent package management. It leverages functional programming concepts to ensure reproducibility and reliability, using the GNU Guile [98] programming language for its core daemon, package definitions and system configurations (L. Courtès [99]).

Central to Guix's philosophy is the concept of reproducible builds and environments. This ensures that software can be built in a deterministic manner, enabling exact replication of software environments at any point in space and time. Guix achieves this by capturing all dependencies, including the toolchain and libraries, in a way that they can be precisely recreated. It supports transactional package upgrades and rollbacks, making system modifications risk-free by allowing users to revert to previous states easily.

Guix uses GNU Guile [98], a Scheme [100] implementation, allowing for more expressive and programmable package definitions. This choice reflects Guix's emphasis on customization and alignment with the Free Software Foundation [101] project's philosophy, rejecting proprietary blobs and aiming for

complete software freedom, which may limit hardware compatibility. Guix's approach can pose a high entry barrier due to its use of a general-purpose functional programming language but offers extensive flexibility for those familiar with Lisp-like languages. That said, users are free to extend Guix with custom packages, free or not.

Guix is committed to ensuring reproducibility and reliability, based on the functional deployment model first introduced by E. Dolstra [102]. It assures reproducible builds by treating software environments as immutable entities, thereby minimising variability across different systems. Guix's approach to software building and package management, grounded in the principles of functional programming and transactional package upgrades, places a strong emphasis on reproducibility. However, this functional paradigm (Definition 15) introduces a learning curve and necessitates a shift from traditional imperative package management methods. Additionally, the adoption of Guix might be further complicated by the absence of non-free software availability, marking a significant consideration for teams considering Guix.

```
(use-modules (guix)
             (guix build-system gnu))

(define-public datetime
  (package
    (name "datetime")
    (version "1.0")
  (source (local-file "./src" #:recursive? #t))
    (build-system gnu-build-system)
    (arguments
     '(
        #:tests? #f
        #:phases
       (modify-phases %standard-phases
         (delete 'configure)
         (replace 'build
           (lambda _ (invoke "gcc" "datetime.c" "-o" "datetime")))
         (replace 'install
           (lambda* (#:key outputs #:allow-other-keys)
             (let ((out (assoc-ref outputs "out")))
               (install-file "datetime" (string-append out "/bin"))))))))
    (synopsis "DateTime Program")
    (description "This package contains a simple program that shows the current date and time.")
    (home-page #f)
    (license #f)))

datetime
```

Listing 14: From Scenario 3, the Guix build file (`guix.scm`)

```
$ cd lib/scenario-3
$ make -s build
/gnu/store/d3gnj6zakm7kwfcrzfdbjxqq41j0x3j1-datetime-1.0
$ make -s check
sha256-iOGz7Lg3BlQLjIi+RMCSqhL6c1UHPLg4U/W0HSQJsJA=
$ make -s run
Built the Jan  1 1970 at 00:00:01.
$ make -s build
/gnu/store/d3gnj6zakm7kwfcrzfdbjxqq41j0x3j1-datetime-1.0
$ make -s check
sha256-iOGz7Lg3BlQLjIi+RMCSqhL6c1UHPLg4U/W0HSQJsJA=
$ make -s run
Built the Jan  1 1970 at 00:00:01.
```

Terminal session 6: Building the C sourcecode from the Guix build file of Scenario 3

3.4.1. Reproducibility In time

In Terminal session 6, we notice on lines 5 and 11 that the output hashes are the same. This is therefore classified as reproducible in time.

3.4.2. Reproducibility In Space

Building the program in a different environment with the same architecture (`x86_64-linux`) resulted in identical output. Compiling the source code on another architecture (`aarch64_darwin`) also produced consistent results, though different from those obtained on `x86_64-linux`. Therefore, we can conclude that the program is reproducible across different environments, *modulo* the hardware architecture.

3.4.3. Reproducibility Of The Build Environment

The reproducibility of the build environment is heavily controlled when using Guix. The dependencies are locked and pinned, it is simply not possible to create a different build environment.

3.5. Evaluation 4 - Nix

Nix [103] is a revolutionary package management system that dramatically reshapes the landscape of software construction, consumption, deployment and management. Its distinctive methodology, grounded in the principles introduced in E. Dolstra [102], marked its inception, setting a new standard for handling software packages. Central to Nix's core is its use of the Nix language, a domain specific Turing-complete language that facilitates the description of software packages, their dependencies, and the environments in which they operate.

The term "Turing-complete" is named after the British mathematician and logician Alan Turing, who introduced the concept of a Turing machine as a fundamental model of computation. A Turing-complete language is a programming language that can simulate a Turing machine, a theoretical device that can solve any computation that can be described algorithmically. Turing completeness is a fundamental property of any programming language that can perform any computation that a Turing machine can, given enough time and memory. This property allows a language to express any algorithm or computation, making it a powerful tool for software development. Examples of Turing-complete languages include: Python, PHP, C++ and JavaScript. On the other hand, non-Turing-complete languages, which are limited in their computational capabilities, include: SQL, Regex and HTML.

This language enables Nix to implement a functional deployment model, ensuring reproducibility, reliability, and portability across different systems by treating packages as functions of their inputs, which results in deterministic builds.

Nix emphasises a deterministic build environment, allowing developers to specify and isolate dependencies explicitly. This method significantly mitigates *"it works on my machine"* issues by providing a high degree of control over the build environment. Nix's strength in ensuring reproducibility comes with the need to embrace its unique approach to system configuration and package management, representing a paradigm shift for new users.

Nix essentially modifies the POSIX standard by installing software in unique locations rather than following the shared file structure described by the Filesystem Hierarchy Standard (FHS). This seemingly minor change brings about several advantageous properties, such as software composition, immutability, configuration rollback, caching and reproducibility.

Nix provides two principal methodologies that are not mutually exclusive: the legacy method (±2006) and the relatively newer *Flake* (±2020) approaches.

3.5.1. Nix legacy method

The legacy way of using Nix involves defining a `default.nix` file that is similar to a function definition in the Nix programming language. This file contains a set of inputs, specifies dependencies, the build command and its output. By default, this method does not enable pure evaluation mode, meaning the hermeticity of the build process is not guaranteed. As a result, potential uncontrolled side effects may occur during the build process. For instance, as demonstrated in Listing 15 at line 2, we manually enforce a very specific version of the `pkgs` variable, a specific snapshot of the Nix package repository that fixes the versions of all packages and libraries. Similarly to the process outlined in Chapter 3, section 3.3 for Docker, this approach, known as "dependency pinning," ensures consistency and reproducibility in the build environment.

```nix
{
  pkgs ? import (fetchTarball "https://github.com/NixOS/nixpkgs/archive/7872526e9c5332274ea5932a0c3270d6e4724f3b.tar.gz") { }
}:

pkgs.stdenv.mkDerivation {
  name = "datetime";

  src = ./src;

  buildPhase = ''
    $CC datetime.c -o datetime
  '';

  installPhase = ''
    install -D datetime $out/bin/datetime
  '';
}
```

Listing 15: The Nix build file (`default.nix`) from Scenario 4

```
$ cd lib/scenario-4
$ make -s build
/nix/store/kr0wn1hsxj0hx86ybxbqda25zza1805p-datetime
$ make -s check
sha256-UiRzZFmePpkBYnAFZe9NeKRb1h1XUJ80OBxv9fRPQCQ=
$ make -s run
Built the Jan  1 1980 at 00:00:00.
$ make -s build
/nix/store/kr0wn1hsxj0hx86ybxbqda25zza1805p-datetime
$ make -s check
sha256-UiRzZFmePpkBYnAFZe9NeKRb1h1XUJ80OBxv9fRPQCQ=
$ make -s run
Built the Jan  1 1980 at 00:00:00.
```

Terminal session 7: Building the C sourcecode with Nix in Scenario 4

3.5.2. Nix Flake

Nix *Flake* introduces a structured approach to managing Nix projects, focusing on reproducibility and ease of use. Currently in an experimental phase, Flake is anticipated to transition to a stable feature soon due to increasing community endorsement (Figure 15) and the tangible reproducibility advantages it offers.

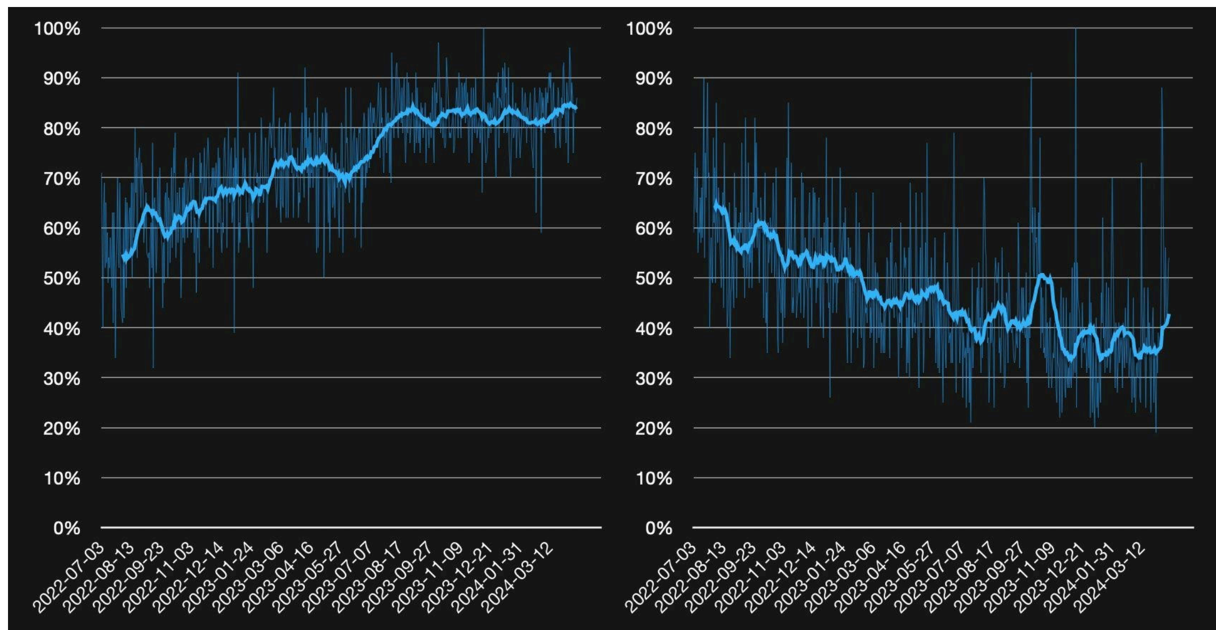


Figure 15: On the left, new repositories containing a `flake.nix` file, and on the right, containing a `default.nix` file (Determinate System)

Flakes aim to simplify and enhance the Nix experience by providing an immutable, version-controlled way to manage packages, resulting in significant improvements in reproducibility and build isolation. Flakes manage project dependencies through a single, top-level `flake.lock` file, which is automatically generated to precisely pin the versions of all dependencies, including transitive ones, as specified in the `flake.nix` file. This file ensures project consistency and reproducibility across different environments.

In addition to altering the Nix command-line syntax, Flakes enforce a specific structure and entry point for Nix expressions, standardising project setup and evaluation. They enable pure evaluation mode by default, which enhances the purity and isolation of evaluations, making builds more consistent and reducing side effects. For instance, making external requests during a build is not possible with Flakes, ensuring that every dependency must be explicitly declared. Flakes require changes to be tracked through `git`, enabling the exact reversion of the project to be pinned in the `flake.lock` file.

The files `flake.nix` and `flake.lock` are complementary and central to the locking mechanism that ensures reproducibility. Together, when committed in a project, they guarantee that every user of a Flake, regardless of when they build or deploy the project, will use the exact same versions of dependencies, thereby ensuring that the project is built consistently every time. However, it is possible to have only a `flake.nix` file without a `flake.lock` file. In such cases, having a reproducible build environment is not guaranteed since dependencies could drift to newer versions.

```nix
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
    flake-parts.url = "github:hercules-ci/flake-parts";
  };

  outputs = inputs @ { flake-parts, ... }: flake-parts.lib.mkFlake { inherit inputs; } {
    systems = [ "x86_64-linux" "x86_64-darwin" "aarch64-darwin" ];

    perSystem = { pkgs, ... }: {
      packages.default = pkgs.stdenv.mkDerivation {
        name = "datetime";

        src = ./src;

        buildPhase = ''
          $CC datetime.c -o datetime
        '';

        installPhase = ''
          install -D datetime $out/bin/datetime
        '';
      };
    };

  };
}
```

Listing 16: The Nix Flake file (`flake.nix`) from Scenario 5

```
$ cd lib/scenario-5
$ make -s build
$ make -s check
sha256-UiRzZFmePpkBYnAFZe9NeKRb1h1XUJ80OBxv9fRPQCQ=
$ make -s run
Built the Jan  1 1980 at 00:00:00.
$ make -s build
$ make -s check
sha256-UiRzZFmePpkBYnAFZe9NeKRb1h1XUJ80OBxv9fRPQCQ=
$ make -s run
Built the Jan  1 1980 at 00:00:00.
```

Terminal session 8: Building the C sourcecode with Nix flake in Scenario 5

3.5.3. Reproducibility In Time

In Terminal session 7, we notice on line 5 and 11 that building twice the sourcecode using Nix's legacy method produces the same output. In Terminal session 8, on line 4 and 9 we notice the same thing. This is therefore classified as reproducible in time.

3.5.4. Reproducibility In Space

Just like Guix, building the program in a different environment with the same architecture (`x86_64-linux`) resulted in identical output. Compiling the source code on another architecture (`aarch64_darwin`) also produced consistent results, though different from those obtained on `x86_64-linux`. Therefore, we can conclude that the program is reproducible across different environments, *modulo* the hardware architecture.

3.5.5. Reproducibility Of The Build Environment

The reproducibility of the build environment is heavily controlled. The dependencies are locked and pinned, it is simply not possible to create a different build environment.

3.5.6. Dealing With Variability

This section will focus on how Nix deals with unstable outputs, highlighting how they have abstracted this issue behind the scenes. The scenarios that will be used are:

- Scenario 6: Building an OCI image with Nix
- Scenario 7: Compiling a Typst document tp a PDF file
- Scenario 8: Compiling a Typst document to a PDF file with Nix, showing how Nix abstracts the issue of non-deterministic builds.
- Scenario 9: Compiling a Typst document with Nix, fixing the issue of non-deterministic builds.

Typst [51] is an advanced markup-based typesetting language that compiles to PDF or SVG. It was initiated in 2019 at the Technical University of Berlin by Laurenz Mädje and Martin Haug. Developed in Rust, this programmable markup language for typesetting became the subject of their master's theses, which they wrote in 2022. After several years of closed-source development, Typst was open-sourced and released to the public in 2023. Despite being relatively recent and lacking a stable version, Typst's maturity has allowed it to be used for writing this master's thesis.

Building OCI images using Docker is a common use case in the software development process. However, the output of the build can be non-deterministic due to the nature of the build process. In scenario 6, we will build an OCI image using Nix only.

```nix
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
    flake-parts.url = "github:hercules-ci/flake-parts";
  };

  outputs = inputs @ { flake-parts, ... }: flake-parts.lib.mkFlake { inherit inputs; } {
    systems = [ "x86_64-linux" "x86_64-darwin" "aarch64-darwin" ];

    perSystem = { pkgs, ... }: {
      packages.default =
        let
          datetime = pkgs.stdenv.mkDerivation {
            name = "datetime";

            src = ./src;

            buildPhase = ''
              $CC datetime.c -o datetime
            '';

            installPhase = ''
              install -D datetime $out/bin/datetime
            '';
          };
        in
        pkgs.dockerTools.buildLayeredImage {
          name = "datetime";
          tag = "latest";
          contents = [ datetime ];
          config.Cmd = [ "/datetime" ];
        };
    };
  };
}
```

Listing 17: The Nix Flake file (`flake.nix`) to build an OCI image in Scenario 6

```
$ cd lib/scenario-6
$ make -s build
load image: datetime
$ make -s check
sha256-dJ3nqJW2bsnSnAca1tOivmWEbk6005mu14N3soM1xxQ=
$ make -s run
Built the Jan  1 1980 at 00:00:00.
$ make -s build
load image: datetime
$ make -s check
sha256-dJ3nqJW2bsnSnAca1tOivmWEbk6005mu14N3soM1xxQ=
$ make -s run
Built the Jan  1 1980 at 00:00:00.
```

Terminal session 9: Building an OCI image with Nix

In Terminal session 9, line 5 and 11, we notice that building twice an OCI image using Nix produces the same output. The Flake file in Listing 17 shows that it is possible to create reproducible OCI containers with Nix, in a simple and declarative way.

In scenario 7, we will compile a trivial Typst document.

Consider the following Typst document on the left, and it's rendering on the right:

```
#set page(width: 10cm, height: auto)
#set heading(numbering: "1.")

= Fibonacci sequence
The Fibonacci sequence is defined through the
recurrence relation $F_n = F_(n-1) + F_(n-2)$.
It can also be expressed in _closed form:_

$ F_n = round(1 / sqrt(5) phi.alt^n), quad phi.alt
= (1 + sqrt(5)) / 2 $

#let count = 8
#let nums = range(1, count + 1)
#let fib(n) = (
  if n <= 2 { 1 }
  else { fib(n - 1) + fib(n - 2) }
)

The first #count numbers of the sequence are:

#align(center, table(
  columns: count,
  ..nums.map(n => $F_#n$),
  ..nums.map(n => str(fib(n))),
))
```

Listing 18: Typst document

1. Fibonacci sequence

The Fibonacci sequence is defined through the recurrence relation $F_n = F_{n-1} + F_{n-2}$. It can also be expressed in *closed form:*

$$F_n = \left\lfloor \frac{1}{\sqrt{5}} \phi^n \right\rceil, \quad \phi = \frac{1 + \sqrt{5}}{2}$$

The first 8 numbers of the sequence are:

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
1	1	2	3	5	8	13	21

Figure 16: Rendering of the Typst document

Terminal session 10 shows that manually compiling the same document twice yields different resulting files.

```
$ cd lib/scenario-7
$ make -s build
$ make -s check
sha256-aOmj1feG5+7h8BDz+URLxr82DxZVrOakDR3B0ULW41c=
$ make -s build
$ make -s check
sha256-CBeUSxEHXqHx0bsERt9cg8hO9qRR/qp2DS9/SrPTtMs=
```

Terminal session 10: Manually compiling a Typst document to a PDF document in Scenario 7

While viewing the resulting PDF files side by side, we notice that they appear totally identical to Figure 16. However, the checksum of those files are different. This discrepancy is common, where the same input can produce different outputs due to non-deterministic behaviour in the build process. Even if the resulting outputs are identical, there can be internal differences. Therefore, given an arbitrary build output, it is impossible to determine if a build is valid or not. It is important to acknowledge that tools like Guix or Nix address this issue by ensuring that the build environment only is consistent and reproducible. In Listing 19, we will show how to compile the same Typst document using Nix and how to eventually fix the discrepancy.

```nix
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
    flake-parts.url = "github:hercules-ci/flake-parts";
  };

  outputs = inputs @ { flake-parts, ... }: flake-parts.lib.mkFlake { inherit inputs; } {
    systems = [ "x86_64-linux" "x86_64-darwin" "aarch64-darwin" ];

    perSystem = { pkgs, ... }: {
      packages.default = pkgs.stdenv.mkDerivation {
        name = "typst-hello-world";

        src = ./src;

        nativeBuildInputs = [ pkgs.typst ];

        buildPhase = ''
          typst compile hello-world.typst hello-world.pdf
        '';

        installPhase = ''
          install -D hello-world.pdf $out/hello-world.pdf
        '';
      };
    };
  };
}
```

Listing 19: The Nix `flake.nix` file to build a Typst document to a PDF in Scenario 8

Compile it twice and observe the outcome:

```
$ cd lib/scenario-8
$ make -s build
$ make -s check
sha256-7c79JDFrrC+MJJd2WavQD1kCsM9wh+knRUOOdAeqMyk=
$ make -s build
$ make -s check
sha256-aq9cxJiiRUxBqxGGV1JjR6+m2JYR7M9h/90R8GGnyxs=
```

Terminal session 11: Building a Typst document in Scenario 8

At lines 4 and 7 of Terminal session 11, we notice that compiling twice a Typst document with Nix produces two different PDF files, their respective checksums are different. While the visual output appears identical, the underlying files are not. At line 3 of Terminal session 12, we leverage a command with specific flags to verify if a build output is reproducible.

```
$ cd lib/scenario-8
$ nix build
$ nix build --rebuild --keep-failed
note: keeping build directory '/tmp/nix-build-typst-hello-world.drv-0'
error: derivation '/nix/store/bsg67bk2csqbjbkib6jfqjqxyncq3l2q-typst-hello-world.drv' may not be
deterministic: output '/nix/store/1n4g9gsq34xzmg351r7sa87rrcazh8gf-typst-hello-world' differs from '/nix/
store/1n4g9gsq34xzmg351r7sa87rrcazh8gf-typst-hello-world.check'
```

Terminal session 12: Checking if a build output is reproducible

Nix will build the document once (line 2), then a second time (line 3) and then compare the output hashes. Thanks to the `--keep-failed` argument, we inform Nix to keep the failed builds so we can do a more introspective analysis of the issue and try to find the root cause of the discrepancy, for example, using `diffoscope` [89] in Terminal session 13.

```
$ diffoscope /nix/store/1n4g9gsq34xzmg351r7sa87rrcazh8gf-typst-hello-world \
            /nix/store/1n4g9gsq34xzmg351r7sa87rrcazh8gf-typst-hello-world.check
```

Terminal session 13: Checking discrepancies between two builds using `diffoscope`

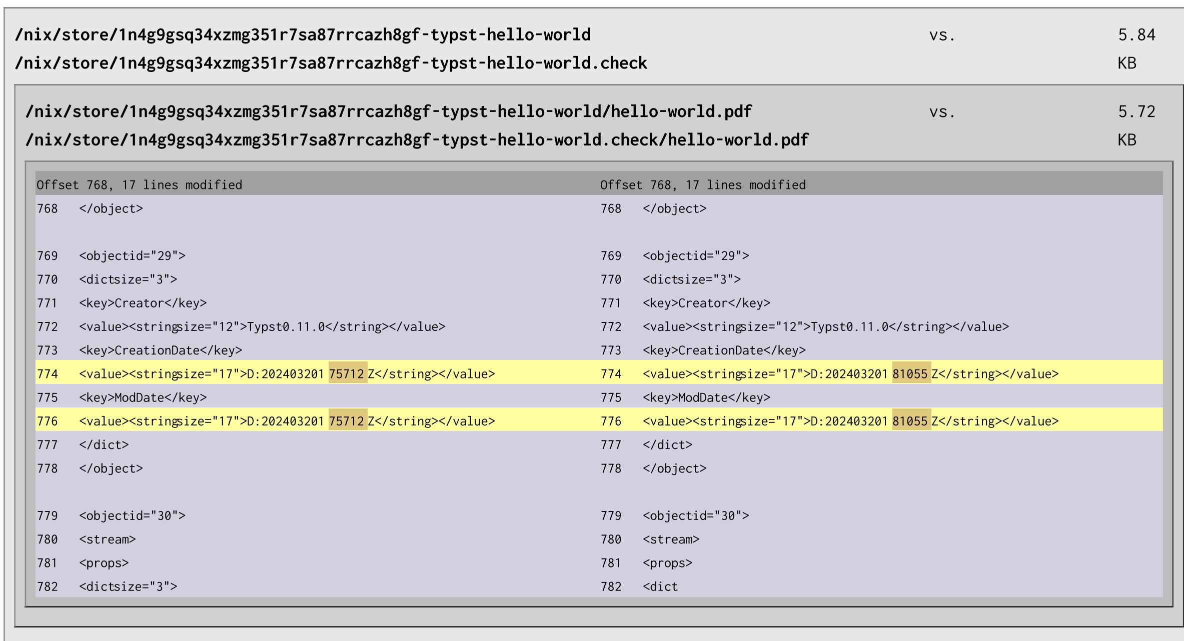


Figure 17: A visual comparison with `diffoscope` of two PDF files generated from the same Typst document

`diffoscope` visually compares the discrepancy between the two PDF files. From the report in Figure 17, the highlighted difference seems to be the creation date metadata. Doing a quick search on Typst Documentation [104] confirms that Typst is able to change the creation date of the output file. Listing 20 implements the trivial change at line 1:

```
#set document(date: none)
#set page(width: 10cm, height: auto)
#set heading(numbering: "1.")

= Fibonacci sequence
The Fibonacci sequence is defined through the
recurrence relation $F_n = F_(n-1) + F_(n-2)$.
It can also be expressed in _closed form:_

$ F_n = round(1 / sqrt(5) phi.alt^n), quad phi.alt = (1 + sqrt(5)) / 2 $

#let count = 8
#let nums = range(1, count + 1)
#let fib(n) = (
  if n <= 2 { 1 }
  else { fib(n - 1) + fib(n - 2) }
)

The first #count numbers of the sequence are:

#align(center, table(
  columns: count,
  ..nums.map(n => $F_#n$),
  ..nums.map(n => str(fib(n))),
))
```

Listing 20: On line 1, the Typst document date is now set to `none`

```
$ cd lib/scenario-9
$ nix build
$ nix build --rebuild --keep-failed
$ # No more error message - the build is fully reproducible now.
```

Terminal session 14: Checking if compiled Typst document is reproducible in Scenario 9

Now we notice that running the command to check if the output is reproducible returns nothing, meaning that the output is fully reproducible.

Often, raising an issue with the upstream project is the most effective method for informing the authors about a problem and monitoring its resolution. In the case of Typst, an issue [52] was documented to describe the problem, and in less than two weeks, it had been addressed and resolved. Consequently, the discrepancy in Terminal session 11 is no longer applicable for Typst versions newer than `0.11.0`.

3.6. Conclusion

In this concluding section of the chapter, a summary of the reproducibility assessment can be found in Table 4. Following the table, this section provides a detailed explanation of our categorization process, outlining the specific criteria used for classifying. Each classification is justified based on the results obtained from our comprehensive empirical evaluation process.

Table 4: Software evaluation comparison

<table>
<tr><th rowspan="2"></th><th rowspan="2">Pro</th><th rowspan="2">Cons</th><th colspan="3">Reproducible within the same hardware architecture</th></tr>
<tr><th>In space</th><th>In time</th><th>Environment</th></tr>
<tr><td>1. Bare compilation</td><td>• Full control over compilation
• Direct understanding of dependencies inherited from host system</td><td>• Prone to “it works on my machine” issues
• Lacks isolation and dependency management</td><td>×</td><td>×</td><td>×</td></tr>
<tr><td>2. Docker</td><td>• Better isolation and dependency management thanks to containerization
• Isolation from host system
• Popular solution, widely adopted</td><td>• Potential variability due to base images and package management
• Additional layer of abstraction due to containerization</td><td>~</td><td>~</td><td>~</td></tr>
<tr><td>3. Guix</td><td>• Deterministic builds with explicit dependency specification
• Functional package management
• Immutable software environments
• Isolation and environment reproducibility
• No containerization overhead</td><td>• Steep learning curve
• Paradigm shift from traditional package management systems required
• Very limited package availability
• Unfree packages are not officially allowed</td><td rowspan="2">✓</td><td rowspan="2">✓</td><td rowspan="2">✓</td></tr>
<tr><td>4. Nix</td><td>• Deterministic builds with explicit dependency specification
• Functional package management
• Immutable software environments
• Isolation and environment reproducibility
• No containerization overhead
• Vast repository of packages, unfree packages are authorized.</td><td>• Steep learning curve
• Pradigm shift from traditional package management systems required</td></tr>
</table>

Legend: ✓ = Supported, ~ = Partially supported, × = Not supported

In evaluating the reproducibility of various tools and methodologies within, a particular focus has been set on the bare compilation method (Chapter 3, section 2). This approach, characterised by its reliance on the host operating system's installed software for compiling source code into executable programs, presents a nuanced challenge to reproducibility. Theoretically, bare compilation allows for a straightforward reproduction of computational results, assuming a static and uniform environment across different computational setups. However, the practical application of this method exposes inherent vulnerabilities to environmental variability. The reliance on the host's installed software means that the exact version of compilers, libraries, and other dependencies can significantly impact the outcome

of the compilation process. These elements are seldom identical across different systems or even over time on the same system, given updates and changes to the software environment. Consequently, the reproducibility promised by Bare compilation is compromised by these external variables, which are often not documented with sufficient rigor or are outside the user's control. Acknowledging these challenges, we categorise the bare compilation (Chapter 3, section 2) as non-reproducible by default, reflecting a practical assessment rather than a theoretical limitation. The classification underscores the significant effort required to document and manage the dependencies on the host's software to achieve a reproducible build process. This perspective is supported by the literature [64], which advocates for standardising and simplifying the management of computational research artefacts. The classification of the method 1 (Chapter 3, section 2) as **non-reproducible** is a pragmatic acknowledgment of the difficulties presented by the dependency on the computational environment.

Docker and similar containerization technologies (Chapter 3, section 3) can facilitate reproducible environments. The reason is that while they provide a high degree of isolation from the host system, they are still subject to variability due to the base images and package managers used within the containers. This variability, however, can be effectively managed with low effort. By meticulously selecting and managing base images and dependencies, it is indeed feasible to elevate Docker from partially to fully reproducible. For these reasons, they are categorised as **partially reproducible**.

Nix (Chapter 3, section 4) and Guix (Chapter 3, section 5) provide a high level of control over the build environment and dependencies, facilitating deterministic and reproducible builds across different systems. By capturing all dependencies and environment specifics in a declarative manner, Nix and Guix offer a reliable and transparent approach to software development. The functional deployment model implemented by Guix, Nix and their forks (like Lix [105]), along with their transactional package upgrades and rollbacks, further enhances reproducibility by enabling exact replication of software environments within the same architecture at any point in space and time.Under the hood, they introduces a novel approach to addressing the challenges of reproducibility. By using a very specific storage model, they ensures that the resulting output directory is determined by the hash of all inputs. This model, while not guaranteeing bitwise identical binaries across all scenarios, especially across different hardware architectures, ensures that the process and environment for building the software are reproducible. Nix and Guix's model represents a significant step forward in mitigating reproducibility challenges within SE. By ensuring that every build can be traced back to its exact dependencies and build environment, it enhances the reliability of software deployments. This approach is particularly beneficial in CI/CD pipelines, where consistency and reliability are paramount. Achieving reproducibility in SE is filled with challenges, from architecture dependencies to non-determinism in compilers. These solutions offers a compelling solution by ensuring reproducible build environments. The exploration of the concepts used in Guix and Nix, and its methodologies provides valuable insights into the complexities of software reproducibility and the necessity for continued research and development in this field. They both are categorised as **reproducible**.

Chapter IV

Conclusion

Reproducibility has the potential to serve as a minimum standard for judging scientific claims when full independent replication of a study is not possible

— R. D. Peng [106]

4.1. Summary

This thesis embarked on an in-depth exploration of reproducibility in SE, motivated by the growing necessity for reliable and repeatable results in research and software development. In this chapter, we summarise our findings and discuss the implications of our research. We also suggest future work that could be done to improve reproducibility.

In the introductory Chapter 1, I provided an overview of my personal journey and the experiences that have led me to pursue this specific area within SE. This chapter offered readers contextual information about my background, especially highlighting my active involvement in open-source communities and my dedicated advocacy for the adoption of reproducibility tools and practices. I delved into my motivations for choosing this topic, underscoring the critical importance of reproducibility in SE. The relevance of establishing reliable and repeatable processes to enhance the integrity of software products and to foster a culture of transparency and collaboration. To guide the reader through the thesis, I recall the goals and a structured overview of the document, chapter by chapter.

In Chapter 2, the theoretical foundations of reproducibility were explored, tracing its origins from classical scientific disciplines where it has long been a cornerstone for validating experimental findings and theories. The chapter began with a historical overview, highlighting how reproducibility emerged as a fundamental principle in the natural sciences and the shifts it has undergone with the advent of the digital era. Following this introduction, a comprehensive set of concepts central to understanding reproducibility in SE was presented. Formal definitions for key terms were introduced, establishing a rigorous and foundational basis for subsequent developments. The narrative then extended into the realms of open source and software security, pivotal areas where reproducibility intersects with broader concerns. Open source software, with its ethos of transparency and collaboration, enhances reproducibility by making the source code readily available. This transparency facilitates the verification of software builds and improves security. With the source code accessible and inputs correctly declared, it allows for immediate identification of dependencies and quicker identification of vulnerabilities. The focus shifted to the specific challenges that software engineers and researchers face in achieving reproducible builds. This discussion delved into obstacles such as non-deterministic build processes, variability of environments, and the lack of, to some extent, standard practices for documenting and sharing the necessary details to replicate software builds. By examining these challenges, the groundwork was laid for identifying effective strategies to address them. Lastly, it introduced third-party tools designed to compare build outputs.

In Chapter 3, I systematically explored and assessed different tools and strategies used to build software with an emphasis on their potential to facilitate reproducible results. The chapter presented a comprehensive analysis of four distinct strategies: Bare compilation, Docker, Guix, and Nix. Each strategy was evaluated through multiple objective criteria. This approach allows for a balanced assessment of each method, providing insights into how well they help in achieving reproducible builds. For Bare compilation, the focus was on the traditional approach of building software directly on the system without containerization or virtual environments, highlighting its limitations and potential issues in reproducibility. Next, Docker was evaluated as a popular containerization technology that encapsulates the software environment, aiming to enhance reproducibility by isolating dependencies. Following Docker, I examined Guix and Nix, two functional package managers that offer more granular control over environment configurations. The chapter concluded with a comparative analysis that ranked these strategies according to their reproducibility potential. This evaluation serves as a resource for developers making decisions about which tools and strategies to implement in their projects to enhance reproducibility.

In the remaining sections of this conclusion chapter, we will summarise the key findings and the broader implications of my research. Additionally, I will expand the discussion by exploring other facets of reproducibility. Finally, I propose future research directions aimed at enhancing reproducibility.

4.2. Evaluation Of Tools

Chapter 3 was dedicated to evaluating different tools and strategies for building software with an emphasis on their potential to facilitate reproducible results.

Bare compilation is the traditional approach to building software directly on the system without containerization or virtual environments. This method offers minimal isolation and control over the build

environment, making it challenging to achieve reproducibility. However, it is still widely used in practice due to its simplicity and familiarity, usually with the underlying operating system ecosystem. Debian based Linux distributions are a common choice for this method due to their extensive package repositories and long-term support, therefore, the usage of the package manager `apt` became a standard practice for installing dependencies when one is missing. This imperative method for installing missing dependencies can lead to non-reproducible builds due to the lack of version pinning. The `apt` package manager does not provide snapshots where all the dependencies are frozen at a given state, which makes it challenging to reproduce the exact build environment. However, it has the advantage of being popular and widely supported.

Using Docker for building software is a popular choice due to its increasing popularity and the ease of creating shareable containers through Docker Hub [70]. However, sharing a container as a single OCI file requires a bit more work and is not as straightforward as sharing a Docker image on their dedicated platform Docker Hub. In a way, thanks to Docker, users have been introduced to the concept of containerization, immutability, and to some extent, reproducibility.

While discussing Docker with people, I noticed a common misconception about reproducibility that is worth noting. To illustrate this, let's consider a project shipping builds of their open-source software through Docker images. At each release, they publish a new version of their image. These Docker images are immutable, and users can use and reuse them at will. However, it is simply impossible to reproduce those images themselves from the sources. While this illustrates the Docker leitmotif *"build once, use everywhere"*, it does not demonstrate true reproducibility. The essence of reproducibility lies in our ability to recreate identical copies of these images from the sources on any machine. If something can be reproduced multiple times but yields different results each time, it is not truly reproducible. Similarly, if something is produced only once and is not meant to be reproduced, it is, to some extent, also not reproducible.

In *declarative configuration management* [107], tools such as Docker, Kubernetes [96], and Terraform [108] are used to specify the desired end state of the system rather than the steps to achieve it. For example, a `Dockerfile` describes the final environment for a container, ensuring that the system matches predefined specifications. This method aligns with the *congruent system management* [109] approach, focusing on consistency and predictability. Declarative configurations ensure idempotence, meaning the same configuration can be applied multiple times without altering the system beyond its intended state. This abstraction makes it easier to understand and maintain, as the system determines the necessary actions to achieve the desired state.

In contrast, *imperative configuration management* [107]involves detailing the exact steps required to transition a system from its current state to a desired state, providing granular control over the configuration process. Tools such as Ansible [110], Chef [111], Puppet [112], and Bash scripts exemplify this approach. This method aligns with the *convergent system management* [109] approach, focusing on achieving a goal through a series of specific actions. While imperative configurations allow for complex logic and conditional operations, they can be challenging to maintain due to their non-idempotent nature, meaning the same script can produce different results depending on the system's initial state. This approach requires meticulous management to ensure consistency and repeatability, offering detailed control at the expense of simplicity and predictability.

The expressiveness of imperative tools comes at a significant cost. These tools allow developers to make stronger assumptions about the current state of the system. This creates a strong likelihood of like environments diverging over time in a process known as *configuration drift*. The declarative approach to configuration management reduces the possibility of configuration drift by favoring idempotence, explicit dependency graphs, and maintaining a strong awareness of the current state of the environment [107, p.348].

During the evaluation, I found Docker to be very easy to use. By using a simple declarative syntax to define the build steps, the `Dockerfile` is one step forward into making configurations more explicit and this has contributed a lot to the success of the Docker ecosystem. On a less positive note, using Docker on other platforms than Linux can be challenging and a deal breaker for some users. On non-

Linux systems, Docker relies on a virtual machine to create a Linux environment, which can lead to performance overheads and latency issues. The differences in filesystems between Linux and non-Linux platforms can also result in inconsistencies and unexpected behaviour in container operations. Additionally, networking configurations and capabilities can vary significantly, causing more complex setups and potential connectivity issues. Resource allocation and management can be less efficient on non-Linux platforms due to the intermediary VM layer. On Linux based architectures, the performance are not as good as running the software natively, but it is still acceptable for most use cases. However, while initiatives such as DevContainer [113] are trying to provide a more integrated experience with Visual Studio Code [114], working with and inside a container adds an extra layer of complexity that can be challenging to manage, especially when dealing with networking, storage and security.

Guix has been an interesting tool to evaluate. While the learning curve is steeper than Docker, the benefits are significant. I appreciated the strict and declarative approach to package management, which aligns well with the reproducibility goals. The idea of using an existing general purpose language for declaring packages and configurations is a powerful idea. The community is small but active, however since no proprietary tools are packaged, it can be a challenge for users to find the software they need. There are workarounds existing but it is not advertised by the Guix community which tend to focus and adhere to the free software philosophy [101]. The performance of Guix is great, since no containerization is involved, the software runs natively on the system and accessing storage and network is a breeze. Guix extensively uses `git` [115] for fetching packages and configurations, and the information displayed to the user while running it is very clean and clear.

Nix has been the most interesting approaches to evaluate, technically but also politically. The learning curve is steep, but the benefits are significant. The Nix language while being Turing-complete (Info box 32) has a very specific and limited Domain Specific Language (DSL), it is remarkably clean and powerful, making it highly suitable for managing package builds and configurations. Additionally, the simplicity of the Nix language enhances its efficiency and usability, positioning it, in my own opinion, as one of the best languages for this task.

During the making of this thesis, I contributed to Nix and I really appreciated how easy it is to contribute but also the transparency of the process when it comes to making a change. The Nix community is very large, active and welcoming, and the Nix package repository `nixpkgs` is one of the most active repository on Github [116]. Nix has completely changed the way I think about software management, how I consume software and how I ship software. I wish I had discovered it earlier.

While Nix offers many advantages, it also has some drawbacks, primarily concerning its installer. The Nix installer is a shell script that downloads and installs Nix on the system. However, there are alternative installers available. During my evaluation, I explored these different installers and ultimately chose one developed by a United States-based company founded by Eelco Dolstra, the creator of Nix. Given this connection, I felt confident using their installer. Nevertheless, I found it peculiar to maintain multiple installers for the same software, which I suspect might be a source of confusion for many users, both new and experienced. Could this indicate a deeper governance problem?

Another drawback of Nix is the documentation. While the Nix manual is extensive and well-written, it can be overwhelming for new users. The manual is comprehensive but it lacks a clear and concise structure, examples and a lot of topics are not covered. A lot of energy, initiative and effort are being made and things are slowly moving in the right direction.

While working with Nix the first time, I got introduced to the concept of Nix channels. Nix channels is a mechanism used to distribute and update collections of Nix expressions. These channels provide users with a way to receive updates for Nix packages and configurations. While Nix extensively uses `git`, the unique concept of channels adds an extra cognitive load for users who want to simply upgrade their machines. I preferred the simplicity Guix offers by just using `git` to update the system. When using Flakes, the concept of Nix channels is no longer needed.

Flakes is an experimental feature as of writing. Released in November 2021 with Nix 2.4 [117], Flakes is powerful yet controversial. For example, some companies, including one founded by the creator of Nix, are advocating for the adoption of Flakes, while the Nix community awaits its stabilization. Although these differences do not affect Nix's functionality in the short term, they can be confusing for new users and may

lead to fragmentation within the community. One might ask why such companies are not contributing to the same codebase as the rest of the community, a situation that has already led to some division.

There are currently around 230 committers spread across the globe taking care of the Nix package repository on Github [118]. On June 1st 2023, I've been granted the status of project committer [119]. This status allows me to merge commits, review code, and contribute to the Nix ecosystem in a more direct way. However, I want to clarify that the conclusions of this thesis were not influenced by my role in the Nix project. I have worked to maintain the highest level of objectivity in this document, with only reproducibility as the primary focus.

At the time of writing, the Nix community was facing a significant crisis, leading to the creation of two new forks of Nix, Lix [105] and Aux [120]. This fragmentation is a major concern for the project's future. While a new governance structure is being established, the community remains divided over different installers, the experimental Flake feature, the sponsorships policy and now forks. Although the current situation is not ideal, I am confident that the community will overcome this crisis and continue to provide an excellent tool for the SE community. The controversial Flake feature has attracted many new users to the Nix ecosystem, and the community is growing rapidly. These issues are likely a result of this rapid growth and the initial lack of clear and transparent governance, a problem that needs to be resolved.

To conclude, it is essential to recognise that Nix is the result of extensive research and development, used in production by numerous companies and individuals. The Nix community is dynamic and vibrant, promising a bright future for the project. The core value of Nix lies in its package repository, `nixpkgs`, which hosts thousands of packages readily accessible to any Nix user. Regardless of which Nix variant or fork one chooses, the true asset remains that package recipe repository, likely to be shared across different forks, ensuring consistency and reliability in package management.

I have concluded that the ideas implemented by Nix stand out as the optimal choice. With two decades of maturity and robustness, the Nix ecosystem is, in my view, currently the best concept for implementing reproducibility in SE. I am convinced that Nix, or a similar technology that embraces the same principles (e.g., Guix [97], Lix [105], Aux [120]), has the potential to revolutionise the way software is built, used, audited, deployed and shared.

4.3. Research Findings

During this research, I have discovered in Chapter 3 that the reproducibility of software builds is a multifaceted challenge that requires a combination of skills, tools and strategies to address it effectively. In this section, I present a summary of the key findings and some other facets of reproducibility that I briefly explored, but could be expanded in future research.

4.3.1. At A Glance

Reproducibility in SE refers to the ability to consistently recreate the same software product, with identical functionality, using the same methods and data across different environments and over time. This involves ensuring that the software build process, dependencies, and computational environments are well-documented and standardised.

Table 5: Pros and cons of reproducibility

Pros	Cons
Facilitates collaboration and onboarding: Reproducibility enables easier collaboration among researchers, developers, as they can replicate and extend each other's work more efficiently.	Steep learning curve: Implementing reproducibility practices may require learning new tools and methodologies, which can be time-consuming and challenging.
Transparency and trust: By sharing the methods, data, and tools used in research and development, other collaborators can verify and build upon the work, fostering a culture of openness and collaboration.	Complexity: The process of making software reproducible can be complex, especially if it hasn't been setup from the beginning of the project.
Improves software quality: Reproducibility practices help in identifying and fixing bugs, improving the software's overall quality and robustness.	Proliferation of package managers: The existence of too many package managers that are built without reproducibility in mind can add another layer of complexity when trying to build software reproducibly.
Enhanced reliability and validity: Reproducible results provide confidence that findings are accurate and not due to random chance or specific initial conditions of a single experiment.	Factors limiting reproducibility: Factors such as proprietary software, licensing issues, and evolving hardware can pose challenges to achieving full reproducibility.
Security and integrity: Ensuring that software can be reliably rebuilt from its source helps in detecting unauthorized changes, enhancing security, and maintaining the integrity of the software supply chain.	Potential for misuse: Over-reliance on automated reproducibility tools can lead to complacency, where developers might not fully understand the underlying processes and methodologies.
Facilitates troubleshooting and debugging: Reproducible experiments serve as a clear benchmark for comparison, assisting teams in identifying discrepancies, tracing error origins, and incrementally enhancing model performance.	Potential for stagnation: Emphasis on reproducibility might slow down innovation as developers might spend more time ensuring reproducibility rather than exploring new ideas and methodologies.

4.3.2. Limitations

Each of the tools evaluated in Chapter 3 offers unique advantages and features, yet they also possess limitations that can impact their effectiveness in achieving reproducibility. Below are some key points noted during the evaluation.

4.3.2.1. Cross Architecture Reproducibility

Achieving software build reproducibility across different operating systems and architectures is not feasible for certain types of build outputs, typically binaries (Chapter 2, section 2.6). Binary build outputs depend on the CPU architecture (e.g., `x86`, `ARM`) because converting source code into machine code uses a set of CPU instructions directly inherited from the CPU architecture. Therefore, obtaining binaries that are identical across every architecture is unlikely to occur. This inherent dependency means that the binaries produced on different architectures will have variations, making exact cross hardware architecture reproducibility unattainable.

4.3.2.2. Unavailability Of Upstream Packages

To illustrate this limitation, I will use the example of a software package that has been removed from the upstream repository and is no longer available for download. In this scenario, the software package cannot be built, as the source code is no longer accessible. Furthermore, any other packages that depend on this now-unavailable package also become impossible to build. This limitation highlights the importance of maintaining a robust and reliable infrastructure for software repositories to ensure the longevity of software packages and facilitate reproducibility. In the absence of such infrastructure, reproducibility becomes increasingly challenging, as software packages may become obsolete or unavailable over time.

To circumvent this limitation, researchers and developers can adopt proactive measures to ensure the reproducibility of their software builds. One approach is to archive the source code and dependencies of the software package, preserving them in a secure and accessible repository. This is what projects like Software Heritage [121] is trying to achieve. By archiving the source code and dependencies, researchers and developers can safeguard against the loss of critical software components and maintain the reproducibility of their builds over time. Additionally, implementing a caching layer to store build outputs can significantly enhance reproducibility. This allows users to retrieve precompiled build outputs, thereby avoiding the need to compile the source code on their machines if the corresponding cached build exists. Nix facilitates the creation of such cached build layers due to its principles (Definition 15), as it produces immutable directories based on sources. This means that modifying existing cached builds is not possible, mitigating potential security issues related to accidental modifications. It's worth noting that this level of immutability and reproducibility is not the case with all package managers.

4.3.2.3. Standardisation

Another limitation is the lack of standard practices for documenting and sharing the necessary details to replicate software builds. This limitation underscores the need for clear and comprehensive documentation to facilitate reproducibility. An exemplary initiative addressing this challenge is the Package URL (PURL) project, which seeks to standardise the identification and location of software packages across various ecosystems and tools. The PURL specification [12] provides a universal syntax to reliably reference software packages, thereby mitigating the inconsistencies that arise from diverse package management conventions. By establishing a common framework, PURL enhances the interoperability and reproducibility of software builds across different platforms and tools, illustrating the critical role of standardisation in reproducible research and development. In addition to PURL, the SWHID is another significant development aimed at improving standardisation. The SWHID provides a unique, persistent identifier for software source code, facilitating the precise identification and retrieval of specific software versions from the Software Heritage Archive [122].

To illustrate this, the 11 June 2024, GitHub announced [123], that generated SBOM files will now include a PURL. By including the PURL, GitHub improves the completeness of the SBOM data, helping users in more clearly identifying the packages in their repositories. This new Github feature exemplifies the practical benefits of adopting a standardised specification, as it addresses a critical need in reproducibility by providing the precise identification of software components used in a build, thereby improving transparency and reproducibility in software development through the inclusion of PURLs in GitHub's SBOMs.

4.4. Future Work

The exploration of reproducibility in SE is an ongoing endeavour. As technology advances and new tools emerge, the landscape of reproducibility continually evolves.

A foundational step towards enhancing reproducibility in SE, and by extension, in the broader realm of science, is to raise awareness of its importance from the outset. This can be achieved through educational initiatives, workshops, and seminars that highlight the benefits of reproducibility and provide researchers with the necessary tools and resources. Embedding reproducible practices into the research culture from the beginning will help establish these practices as standard requirements rather than optional enhancements.

On the technical side, the frameworks and tools evaluated in this thesis provide a robust foundation for reproducible software builds. However, significant scope for improvement remains. Currently, various Linux distributions continue to develop their own package managers, resulting in redundant efforts and inefficient use of resources. While this idea seems utopic, adopting Nix as a *universal package manager* could be a potential solution. Nix abstracts away the underlying system, making it an ideal candidate for standardizing software deployment across different Linux distributions. By providing a consistent environment, Nix could streamline the deployment process, reduce inconsistencies, enhance reproducibility across diverse systems, and improve security. With a universal package manager, security vulnerabilities could be addressed more efficiently, as fixes could be deployed across all systems simultaneously. However, implementing Nix universally presents several challenges, such as ensuring compatibility with all distributions, overcoming resistance from communities accustomed to their current systems, and managing the scalability and maintenance of such an initiative. To address these challenges, a phased approach could be adopted, starting with specific use cases or distributions where Nix has demonstrated clear benefits. Additionally, collaborative efforts and open dialogue among stakeholders could facilitate a smoother transition. Furthermore, adopting Nix could significantly reduce our carbon footprint by eliminating the need to store prebuild binaries for different distributions. Instead, binaries would be prebuilt once, then stored and made available on a Content Delivery Network (CDN) for all the Linux distributions, thus streamlining the deployment process and contributing to environmental sustainability.

The complexity of reproducibility is comparable to that of replicating a painting. While explaining this thesis to a painter, I used the example of creating an indistinguishable copy of another painting. The painter explained that the likelihood of achieving such perfect replication is comparable to the chance of a monkey writing Shakespeare's work due to the numerous variables involved, such as the type of paint, the brush, the canvas, the lighting, the environment, the painter's mood, and the time of day. This carefully chosen analogy underscores the multifaceted nature of reproducibility in SE, where numerous variables and intricate interplay influence the final outcome.

The challenges in achieving reproducibility in artistic works highlight the complexity and necessity of considering various factors. This broader context emphasises that reproducibility is not limited to SE but is a universal issue that requires ongoing attention and innovation. Across different fields such as the arts, social sciences, and natural sciences, achieving reproducibility involves addressing a wide array of considerations. Specifically, these considerations include ethical, economic, philosophical and educational aspects.

4.4.1. Flaky tests

Flaky tests are tests that exhibit inconsistent outcomes without changes to the code being tested. This means that they can fail or pass sporadically, leading to uncertainty and mistrust in the test outcomes. These unreliable tests are not only problematic for developers but also hinder the effectiveness of valuable techniques in software testing research. Essentially, flaky tests pose a threat to the validity of methodologies that rely on the assumption that a test's outcome is solely determined by the source code it evaluates. From a recent paper [124], a survey of software developers found that 59% claimed to deal with flaky tests on a monthly, weekly, or daily basis.

Reproducibility is directly linked to the issue of flaky tests because their inconsistency directly impacts the ability to reproduce results reliably. For effective reproducibility in scientific software and other domains, it is crucial that tests yield consistent and predictable results. Unstable tests hinder the verification process, making it difficult to assert whether observed issues are due to actual code defects or just the

flakiness of the tests themselves. This discrepancy affects validation, verification, and the confidence in computational results.

Future work on this issue should focus on developing techniques to identify and mitigate flaky tests, ensuring that test outcomes are consistent and reliable. Research could explore advanced methods for detecting flakiness, such as machine learning algorithms that analyze test behavior patterns. Additionally, creating tools to automatically stabilize flaky tests and integrating these solutions into continuous integration pipelines would significantly enhance the reliability and trustworthiness of software testing processes.

4.4.2. Formal Concepts

There is potential for further describing and refining the formal definitions related to reproducibility introduced mostly in Chapter 2. While this thesis has introduced formal definitions of key terms related to reproducibility, these definitions can be expanded, refined and improved.

Having a set of formal definitions related to reproducibility is important because they provide clarity and consistency in terminology, helping researchers communicate more effectively. Standardised definitions allow for consistent evaluation criteria, making it easier to compare results across different studies and ensuring reliable assessments. This standardisation also supports the development of tools and methodologies for verifying reproducibility, making the evaluation process more rigorous.

Moreover, formal definitions play an educational role by instilling a culture of reproducibility among new researchers and students. They guide policy-making and governance in research institutions, promoting best practices. Ultimately, these definitions ensure the reliability and integrity of scientific findings. Enhancing these formal definitions will further strengthen the quality and credibility of research.

4.4.3. Ethical Considerations

Reproducibility is fundamentally intertwined with ethical practices in research, as it bolsters the scientific process by enforcing transparency among researchers. The commitment to making research reproducible serves the scientific community and fosters public trust in scientific outcomes, demonstrating a respect for the integrity of science and its impact on society.

Ethically, researchers are obliged to report their findings and also provide comprehensive details of their methodologies. This level of accountability allows their research to be rigorously scrutinised, validated, or refuted by peers, thus enhancing the quality and credibility of the scientific knowledge produced. It is essential for maintaining public trust in scientific research. When results are not reproducible, it undermines the reliability of scientific discourse and can lead to scepticism towards scientific claims. This is particularly critical when scientific research informs policy decisions in crucial areas such as public health and environmental conservation, where non-reproducible research could lead to misguided policies with far-reaching consequences. As we've seen through this document, the open sharing of data and methods is a cornerstone of reproducible research. It democratises access to scientific knowledge, enabling a diverse range of researchers to participate in and contribute to scientific discovery, regardless of their geographical, political or institutional affiliations. Reproducibility acts as a bulwark against fraud and bias. It ensures that research content, findings, discoveries are genuine and not the result of manipulated data, thus promoting fair distribution of resources and recognition within the scientific community.

4.4.4. Philosophical Considerations

According to K. R. Popper [18], a cornerstone of scientific inquiry is that a theory must be falsifiable; that is, it can be disproven through empirical evidence. While evidence alone cannot conclusively verify a hypothesis, it can refute one. Reproducibility is essential in this context as it allows hypotheses to be rigorously tested and either validated or refuted, thus contributing to the evolution of scientific truth.

Scientific knowledge is not static but accumulates iteratively. Reproducibility fortifies this process by ensuring that each new discovery builds upon a foundation of previously verified results. This methodological consistency is crucial for the progressive nature of scientific understanding.

Reproducibility also underpins the pursuit of objective knowledge. It helps distinguish robust scientific results from those that are anomalies or artefacts of experimental error, refining our collective understanding of natural phenomena. The establishment of scientific consensus relies heavily on

reproducible results. This reproducibility facilitates agreement among scientists on what constitutes established facts, thus propelling scientific progress and fostering collaboration across various disciplines.

Moreover, reproducibility enhances the scientific enterprise by encouraging the open sharing of data and methods. This openness not only fosters collaboration but also transforms research into a collective endeavour rather than a series of isolated efforts. It cultivates a scientific culture where data transparency and methodological openness are normative, promoting an inclusive environment that respects and builds upon the work of fellow researchers. By facilitating the verification of results, reproducibility pays homage to the foundational work of previous researchers and ensures that their contributions to knowledge are respected and built upon. It reinforces the integrity of scientific practice and propels the pursuit of further inquiry.

4.4.5. Economical Considerations

Reproducibility intersects with economic efficiency. Efficient reproducibility can accelerate scientific progress by enabling quicker validation of results and facilitating broader dissemination of knowledge. Economies of scale can be applied where repetitive reproductions are feasible, thus reducing the unit cost of research and making large-scale studies more financially sustainable.

The economic impact of reproducibility also extends to its utility in policymaking and industrial applications. Reproducible research ensures that policies and commercial ventures based on scientific findings are underpinned by robust and reliable evidence, thus minimising risks and maximising efficacy. This not only bolsters public and investor confidence but also enhances the economic utility of scientific research.

To some extent, reproducibility is closely linked to the economy of scarcity, where the rarity of an object or finding directly impacts its reproducibility and associated costs. Rare phenomena or data require more specialised resources for reproduction, which are often costly and less accessible. This scarcity increases the economic investment required to replicate a study, from securing rare materials to accessing specialised equipment.

Conversely, phenomena that are not rare can be reproduced with greater ease and at a lower cost. The abundance of necessary resources and established methodologies makes such reproduction economically viable and less resource-intensive. This disparity highlights a fundamental economic principle within scientific research: the cost and feasibility of reproducibility often depend on the availability and accessibility of resources.

To illustrate this, consider the manual replication of a painting, where the scarcity of the original artist's brushstrokes and techniques makes it challenging to reproduce the artwork with the same precision and quality. It would take a significant investment of time, effort, and resources to manually replicate the painting accurately. Choosing the proper materials, mastering the techniques, and recreating the environment and artist's vision are all essential factors that contribute to the cost and feasibility of reproducing the painting. In contrast, a mass-produced item, such as a digital photograph, can be replicated with relative ease and at a lower cost.

In summary, the economic implications of reproducibility encompass a range of considerations from the broader economic impacts on efficiency, credibility, and practical application to the costs of rare resources.

4.4.6. Educational Considerations

Educating students in best practices is crucial for fostering a culture of reproducibility. We can draw a compelling parallel with mathematics, where reproducibility is inherently embedded. Just as mathematical proofs and solutions can be independently verified by anyone following the same steps and logic, reproducibility in SE aims for the same level of transparency and verifiability. This bridge underscores that reproducibility is a desirable trait and a foundational principle that should be rigorously applied in computational research. For example, a professor might provide students with the necessary data, software, and materials for a specific course through fully reproducible methods and tools. This could involve using open-source software, version control systems, and detailed documentation to ensure that students can reproduce the outcome in any space and any time.

Incorporating ethics education into research training programmes helps instil the importance of reproducibility and integrity in scientific research, emphasising the ethical responsibility researchers

have towards producing verifiable and reliable results. Organising workshops and seminars focused on reproducibility can help disseminate best practices and foster a community dedicated to high standards in research. These events serve as platforms for discussion, collaboration, and the sharing of new tools and techniques.

Furthermore, experienced researchers mentoring early-career scientists can pass on valuable knowledge and emphasise the importance of reproducibility in their work. Mentorship provides hands-on guidance and support, helping to build a strong foundation for the next generation of scientists. By addressing these educational and training considerations, we can cultivate a research environment that values and prioritises reproducibility.

Reproducibility is closely linked to fact-checking, as both processes involve verifying the accuracy and reliability of research findings. Emphasising reproducibility can significantly enhance the quality of fact-checking by providing clear, transparent methodologies and robust data that others can independently verify. This rigorous approach ensures the credibility of scientific research and fosters critical thinking skills. By engaging in reproducible research practices, individuals develop a keen critical thinking, which is essential for evaluating information, identifying biases, and making informed decisions. Teaching the principles of reproducibility and fact-checking from an early age is crucial. Incorporating these concepts into school curricula helps students develop critical thinking skills early on, empowering them to question assumptions, evaluate evidence, understand the scientific process and verify by themselves. Educating students about the importance of transparency, data integrity, and methodological rigour lays the foundation for a more scientifically literate and critically minded society. By fostering these skills from the beginning of their education, we can equip future generations with the tools they need to navigate the complex and information-rich world.

This page is intentionally left blank.

List of definitions

Definition 1: Experiment 24
Definition 2: Repeatability (Same team, same experimental setup) 24
Definition 3: Reproducibility (Different team, different experimental setup) 25
Definition 4: Replicability (Different team, same experimental setup) 25
Definition 5: Reproducibility 25
Definition 6: Reproducibility at build time 26
Definition 7: Reproducibility at run time 26
Definition 8: Computation 30
Definition 9: inputs and outputs 31
Definition 10: Evaluation 33
Definition 11: Pure function 34
Definition 12: Impure function 34
Definition 13: Reproducibility 36
Definition 14: Deterministic build 41
Definition 15: Functional package management 50

This page is intentionally left blank.

List of figures

Figure 1: Scientific method 23
Figure 2: Scientific method with reproducibility 23
Figure 3: Reproducibility states 27
Figure 4: Inputs, computation, outputs 30
Figure 5: Functions in the context of Mathematics 30
Figure 6: Functions in the context of CS 30
Figure 7: Inputs, computation, outputs 31
Figure 8: The input is now composed of the program and its parameters 32
Figure 9: The input is now composed of the program and its parameters and the environment where it is going to be evalued. 32
Figure 10: The evaluation of inputs into outputs where the input is composed of the program and its parameters and the environment where it is going to be evaluated. 33
Figure 11: Inputs, checksum, string output 35
Figure 12: Dependency graph of `my-app` version `1.2.3`, where a flaw has been detected in `xz` dependency 39
Figure 13: The reproducible builds approach to increasing trust in executables built by untrusted third parties. 40
Figure 14: A `diffoscope` report using HTML format 46
Figure 15: On the left, new repositories containing a `flake.nix` file, and on the right, containing a `default.nix` file (Determinate System) 59
Figure 16: Rendering of the Typst document 62
Figure 17: A visual comparison with `diffoscope` of two PDF files generated from the same Typst document 64

This page is intentionally left blank.

List of tables

Table 1: The four levels of reproducibility and their requirements. 23
Table 2: Classes of reproducibility 32
Table 3: Nuances between functions and computations 37
Table 4: Software evaluation comparison 65
Table 5: Pros and cons of reproducibility 72

This page is intentionally left blank.

Bibliography

[1] A. Tsolomitis, “The New Computer Modern Font CTAN project.” [Online]. Available: https://ctan.org/pkg/newcomputermodern

[2] P. Dellaiera, “Reproducibility in Software Engineering.” [Online]. Available: https://github.com/drupol/master-thesis

[3] “GitHub Actions.” [Online]. Available: https://github.com/features/actions

[4] Zenodo, “Zenodo.” [Online]. Available: https://zenodo.org/

[5] P. Dellaiera, “Reproducibility in Software Engineering.” [Online]. Available: https://github.com/drupol/r13y-build-scenarios/

[6] Creative Commons, “Creative Commons Attribution 4.0 International.” [Online]. Available: https://creativecommons.org/licenses/by/4.0/

[7] European Commission and DG CNECT, *Cyber Resilience Act*. 2022. [Online]. Available: https://ec.europa.eu/newsroom/dae/redirection/document/89543

[8] CycloneDX, “CycloneDX.” [Online]. Available: https://cyclonedx.org/

[9] C. Ada Ehmke, “The Hippocratic License 3.0.” [Online]. Available: https://firstdonoharm.dev/

[10] IEEE, “IEEE website.” [Online]. Available: https://www.ieee.org/

[11] Open Container Initiative, “Open Container Initiative.” [Online]. Available: https://opencontainers.org/

[12] P. Ombredanne, “PURL Specification.” [Online]. Available: https://github.com/package-url/purl-spec

[13] Software Package Data Exchange, “Software Package Data Exchange (SPDX).” [Online]. Available: https://spdx.org/

[14] D. Akhawe, F. Braun, F. Marier, and J. Weinberger, “Subresource Integrity. W3C Recommendation. W3C.” [Online]. Available: https://www.w3.org/TR/SRI/

[15] R. Di Cosmo, M. Gruenpeter, and S. Zacchiroli, “Identifiers for Digital Objects: the Case of Software Source Code Preservation,” in *iPRES 2018 - 15th International Conference on Digital Preservation*, Sep. 2018, pp. 1–9. doi: 10.17605/OSF.IO/KDE56.

[16] T. Preston-Werner, “Semantic Versioning 2.0. 0,” *l\inea]. Available: http://semver. org*, 2013.

[17] R. Bacon, *Opus Majus, Volumes 1 and 2*. Philadelphia: University of Pennsylvania Press, 1928. doi: 10.9783/9781512814064.

[18] K. R. Popper, *The Logic of Scientific Discovery*. London: Hutchinson, 1934.

[19] K. Thompson, “Reflections on Trusting Trust,” *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984, doi: 10.1145/358198.358210.

[20] M. Fourné, D. Wermke, W. Enck, S. Fahl, and Y. Acar, “It’s like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security,” in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1527–1544. doi: 10.1109/SP46215.2023.10179320.

[21] J. F. Claerbout and M. Karrenbach, “Electronic documents give reproducible research a new meaning,” in *SEG Technical Program Expanded Abstracts 1992*, 1992, pp. 601–604. doi: 10.1190/1.1822162.

[22] C. Collberg and T. A. Proebsting, “Repeatability in Computer Systems Research,” *Commun. ACM*, vol. 59, no. 3, pp. 62–69, Feb. 2016, doi: 10.1145/2812803.

[23] R. Alkhadra, J. Abuzaid, M. AlShammari, and N. Mohammad, “Solar Winds Hack: In-Depth Analysis and Countermeasures,” in *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, 2021, pp. 1–7. doi: 10.1109/ICCCNT51525.2021.9579611.

[24] Joinup - Open Source Observatory, “European Commission to use open source messaging service Signal.” [Online]. Available: https://joinup.ec.europa.eu/collection/open-source-observatory-osor/news/signal-messaging-service

[25] Signal Foundation, “Signal, A private messenger.” [Online]. Available: https://https//signal.org

[26] S. Altman, “OpenAI DevDay: Opening Keynote.” [Online]. Available: https://youtu.be/U9mJuUkhUzk?t=421

[27] “Reproducible Signal builds for Android.” [Online]. Available: https://signal.org/blog/reproducible-android/

[28] “Reproducible Builds for iOS and Android.” [Online]. Available: https://telegram.org/blog/verifiable-apps-and-more

[29] E. Dolstra, “NixOS.” [Online]. Available: https://nixos.org/

[30] Github NixOS project, “Pol Dellaiera's pull requests in the Nix project.” [Online]. Available: https://github.com/NixOS/nixpkgs/pulls?page=1&q=is%3Apr+is%3Aclosed+author%3Adrupol

[31] Github NixOS project, “Pol Dellaiera's reviews in the Nix project.” [Online]. Available: https://github.com/NixOS/nixpkgs/pulls?q=is%3Apr+is%3Aclosed+reviewed-by%3Adrupol

[32] Github NixOS project, “Nix builder for PHP applications.” [Online]. Available: https://github.com/NixOS/nixpkgs/pull/225401

[33] Github NixOS project, “Nix builder for PHP applications, version 2.” [Online]. Available: https://github.com/NixOS/nixpkgs/pull/308059

[34] R. Lerdorf, “PHP.” [Online]. Available: https://php.net/

[35] N. Adermann and J. Boggiano, “Composer.” [Online]. Available: https://getcomposer.org/

[36] Github Composer project, “Composer reproducible pull request.” [Online]. Available: https://github.com/composer/composer/pull/11663

[37] International PHP Conference, “Leverage Nix in the PHP ecosystem.” [Online]. Available: https://phpconference.com/web-development/leveraging-nix-php-ecosystem/

[38] Github PHPunit project, “Add composer.lock.” [Online]. Available: https://github.com/sebastianbergmann/phpunit/pull/5576

[39] Github Psysh project, “Consider adding a composer.lock file in the repository.” [Online]. Available: https://github.com/bobthecow/psysh/issues/767

[40] Github GrumPHP project, “Consider adding a composer.lock file in the repository.” [Online]. Available: https://github.com/phpro/grumphp/issues/1095

[41] Github Psalm project, “Adding composer.lock in VCS ?.” [Online]. Available: https://github.com/vimeo/psalm/issues/10446

[42] Github PHPMD project, “Adding composer.lock in VCS ?.” [Online]. Available: https://github.com/phpmd/phpmd/issues/1056

[43] Github PHP-CS-Fixer project, “Publish composer.lock for each release.” [Online]. Available: https://github.com/PHP-CS-Fixer/PHP-CS-Fixer/issues/7590

[44] Github PHP Parallel Lint project, “Adding composer.lock under VCS ?.” [Online]. Available: https://github.com/php-parallel-lint/PHP-Parallel-Lint/issues/153

[45] “Make PHAR reproducible.” [Online]. Available: https://github.com/theseer/Autoload/issues/114

[46] Reproducible Builds, “Reproducible Builds Website.” [Online]. Available: https://reproducible-builds.org/

[47] Reproducible Builds project, “flake: add Nix flake files.” [Online]. Available: https://salsa.debian.org/reproducible-builds/reproducible-website/-/merge_requests/102

[48] Reproducible Builds project, “Add bibliography.bib file to the repository and update Academic Publications page accordingly.” [Online]. Available: https://salsa.debian.org/reproducible-builds/reproducible-website/-/merge_requests/113

[49] Reproducible Builds project, “buy-in: add SBOM and ephemeral development environments.” [Online]. Available: https://salsa.debian.org/reproducible-builds/reproducible-website/-/merge_requests/114

[50] “Git repository of the reproducible-builds.org website.” [Online]. Available: https://salsa.debian.org/reproducible-builds/reproducible-website/-/commits/master?author=Pol%20Dellaiera

[51] L. Mädje, M. Haug, and The Typst Project Developers, “Typst.” [Online]. Available: https://github.com/typst/typst

[52] Github Typst project, “Enhancing reproducibility of Typst documents with SOURCE_DATE_EPOCH.” [Online]. Available: https://github.com/typst/typst/issues/3806

[53] Github Typst project, “More control over typst compilation environment.” [Online]. Available: https://github.com/typst/typst/issues/3892

[54] European Commission, “Devs profile.” [Online]. Available: https://code.europa.eu/ecphp/devs-profile

[55] European Commission, “ECPHP Sessions.” [Online]. Available: https://code.europa.eu/ecphp/sessions

[56] P. Dellaiera, “Summer Of Nix 2022 - State of Nix at European Commission.” [Online]. Available: https://www.youtube.com/watch?v=I7wdcJ3YhoU

[57] T. C. Durand and V. Tapas, “La Tronche En Biais.” [Online]. Available: https://www.youtube.com/user/TroncheEnBiais

[58] T. C. Durand and V. Tapas, “SCIENCES : Une crise de reproductibilité des études ?.” [Online]. Available: https://www.youtube.com/watch?v=OA8Eki7fvAw

[59] J. T. Cacioppo, R. M. Kaplan, J. A. Krosnick, J. L. Olds, and H. Dean, “Social, behavioral, and economic sciences perspectives on robust and reliable science,” 2015.

[60] M. Castillo, “The Scientific Method: A Need for Something Better?,” *American Journal of Neuroradiology*, vol. 34, no. 9, pp. 1669–1671, 2013, doi: 10.3174/ajnr.A3401.

[61] B. T. Essawy *et al.*, “A taxonomy for reproducible and replicable research in environmental modelling,” *Environmental Modelling & Software*, vol. 134, p. 104753–104754, 2020, doi: 10.1016/j.envsoft.2020.104753.

[62] National Academies of Sciences Engineering and Medicine, *Reproducibility and Replicability in Science*. Washington, DC: The National Academies Press, 2019. doi: 10.17226/25303.

[63] L. A. Barba, “Terminologies for Reproducible Research,” *arXiv*, Feb. 2018, doi: 10.48550/arXiv.1802.03311.

[64] M. Schwab, N. Karrenbach, and J. Claerbout, “Making scientific computations reproducible,” *Computing in Science & Engineering*, vol. 2, no. 6, pp. 61–67, 2000, doi: 10.1109/5992.881708.

[65] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden, “Reproducible Research in Computational Harmonic Analysis,” *Computing in Science & Engineering*, vol. 11, no. 1, pp. 8–18, 2009, doi: 10.1109/MCSE.2009.15.

[66] S. N. Goodman, D. Fanelli, and J. P. A. Ioannidis, “What does research reproducibility mean?,” *Science Translational Medicine*, vol. 8, no. 341, p. 341–342, 2016, doi: 10.1126/scitranslmed.aaf5027.

[67] R. D. Peng, “Reproducible research and Biostatistics,” *Biostatistics*, vol. 10, no. 3, pp. 405–408, 2009, doi: 10.1093/biostatistics/kxp014.

[68] Association for Computing Machinery, “Artifact Review and Badging.” [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current

[69] J. Malka, S. Zacchiroli, and T. Zimmermann, “Reproducibility of Build Environments through Space and Time,” in *46th International Conference on Software Engineering (ICSE 2024) - New Ideas and Emerging Results (NIER) Track*, Apr. 2024. doi: 10.48550/arXiv.2402.00424.

[70] Docker, Inc., “Docker Hub.” [Online]. Available: https://hub.docker.com/

[71] L. A. Barba, “Defining the role of open source software in research reproducibility.” [Online]. Available: https://arxiv.org/abs/2204.12564

[72] N. Larigaldie, A. Dreneva, and J. L. Orquin, “eyeScrollR: A software method for reproducible mapping of eye-tracking data from scrollable web pages,” *Behavior Research Methods*, Feb. 2024, doi: 10.3758/s13428-024-02343-1.

[73] K. Hinsen, “Reproducible computations with Guix,” 2020, [Online]. Available: https://guix.gnu.org/en/blog/2020/reproducible-computations-with-guix/

[74] P. Ivie and D. Thain, “Reproducibility in Scientific Computing,” *ACM Comput. Surv.*, vol. 51, no. 3, Jul. 2018, doi: 10.1145/3186266.

[75] L. Courtès, “Building a Secure Software Supply Chain with GNU Guix,” *The Art, Science, and Engineering of Programming*, vol. 7, no. 1, Jun. 2022, doi: 10.22152/programming-journal.org/2023/7/1.

[76] Joe Biden, “Executive Order 14028: Improving the Nation's Cybersecurity.” [Online]. Available: https://www.federalregister.gov/d/2021-10460

[77] J. Malka, “Increasing Trust in the Open Source Supply Chain with Reproducible Builds and Functional Package Management,” in *46th International Conference on Software Engineering (ICSE 2024) - Doctoral Symposium (DS) Track*, Apr. 2024. doi: 10.1145/3639478.3639806.

[78] A. Decan and T. Mens, “What Do Package Dependencies Tell Us About Semantic Versioning?,” *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1226–1240, 2021, doi: 10.1109/TSE.2019.2918315.

[79] National Institute of Standards and Technology, "CVE-2024-3094." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-3094

[80] L. Collin, "XZ Utils." [Online]. Available: https://tukaani.org/xz/

[81] P. Mueller and B. Yadegari, "The stuxnet worm," *Département des sciences de l'informatique, Université de lArizona*, 2012, [Online]. Available: https://www2.cs.arizona.edu/~collberg/Teaching/466-566/2012/Resources/presentations/topic9-final/report.pdf

[82] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, "Heartbleed 101," *IEEE Security & Privacy*, vol. 12, no. 4, pp. 63–67, 2014, doi: 10.1109/MSP.2014.66.

[83] C. Lamb and S. Zacchiroli, "Reproducible Builds: Increasing the Integrity of Software Supply Chains," *CoRR*, 2021, doi: 10.48550/arXiv.2104.06020.

[84] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[85] S. Loosemore, R. Stallman, R. McGrath, A. Oram, and U. Drepper, *The GNU C library reference manual*. 2023. [Online]. Available: https://sourceware.org/glibc/manual/2.39/pdf/libc.pdf

[86] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st ed. USA: No Starch Press, 2010.

[87] Github NixOS project, "stdenv: option to determine SOURCE_DATE_EPOCH in fetchers." [Online]. Available: https://github.com/NixOS/nixpkgs/pull/256270

[88] Strip Nondeterminism, "Strip Nondeterminism." [Online]. Available: https://salsa.debian.org/reproducible-builds/strip-nondeterminism

[89] Reproducible Builds, "diffoscope." [Online]. Available: https://diffoscope.org/

[90] A. C. Clarke, *Profiles of the Future; an Inquiry into the Limits of the Possible*. Harper & Row, 1973.

[91] L. Courtès and R. Wurmus, "Reproducible and User-Controlled Software Environments in HPC with Guix," in *Euro-Par 2015: Parallel Processing Workshops*, Cham: Springer International Publishing, 2015, pp. 579–591. doi: 10.1007/978-3-319-27308-2_47.

[92] A. Decan, T. Mens, P. R. Mazrae, and M. Golzadeh, "On the in Software Development Repositories," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 235–245. doi: 10.1109/ICSME55016.2022.00029.

[93] The Free Software Foundation, "GNU Make." [Online]. Available: https://www.gnu.org/software/make/

[94] Docker, Inc., "Docker." [Online]. Available: https://www.docker.com/

[95] Podman, "Podman." [Online]. Available: https://podman.io/

[96] Kubernetes, "Kubernetes." [Online]. Available: https://kubernetes.io/

[97] Guix, "Guix Website." [Online]. Available: https://guix.gnu.org/

[98] GNU Guile, "GNU Guile." [Online]. Available: https://www.gnu.org/software/guile/

[99] L. Courtès, "Functional Package Management with Guix." 2013. doi: 10.48550/arXiv.1305.4584.

[100] R. K. Dybvig, *The SCHEME programming language*. Mit Press, 2009.

[101] Free Software Foundation, "The Free Software Foundation." [Online]. Available: https://www.fsf.org/

[102] E. Dolstra, “The Purely Functional Software Deployment Model,” 2006. [Online]. Available: https://dspace.library.uu.nl/handle/1874/7540

[103] Nix, “Nix.” [Online]. Available: https://nixos.org/

[104] Typst Documentation, “Typst documentation website.” [Online]. Available: https://typst.app/docs/

[105] Lix, “Lix.” [Online]. Available: https://lix.systems/

[106] R. D. Peng, “Reproducible Research in Computational Science,” *Science*, vol. 334, no. 6060, pp. 1226–1227, 2011, doi: 10.1126/science.1213847.

[107] T. Hunter and S. Porter, *Google Cloud Platform for Developers*. Birmingham, England: Packt Publishing, 2018.

[108] HashiCorp, “Terraform.” [Online]. Available: https://www.terraform.io/

[109] S. Traugott, “Why Order Matters: Turing Equivalence in Automated Systems Administration.,” 2002, pp. 99–120.

[110] Ansible, “Ansible.” [Online]. Available: https://www.ansible.com/

[111] Chef, “Chef.” [Online]. Available: https://www.chef.io/

[112] Puppet, “Puppet.” [Online]. Available: https://puppet.com/

[113] Visual Studio Code, “Developing inside a Container.” [Online]. Available: https://containers.dev/

[114] Microsoft, “Visual Studio Code.” [Online]. Available: https://code.visualstudio.com/

[115] L. Torvalds, “Git.” [Online]. Available: https://git-scm.com/

[116] GitHub, “The State of the Octoverse 2022.” [Online]. Available: https://octoverse.github.com/2022/state-of-open-source

[117] E. Dolstra, “Nix 2.4 release announcement.” [Online]. Available: https://discourse.nixos.org/t/nix-2-4-released/15822

[118] Nixpkgs, “Nixpkgs committers.” [Online]. Available: https://github.com/orgs/NixOS/teams/nixpkgs-committers

[119] P. Dellaiera, “Nixpkgs committer grant.” [Online]. Available: https://github.com/NixOS/nixpkgs/issues/50105#issuecomment-1571885173

[120] Aux, “Aux.” [Online]. Available: https://aux.computer/

[121] R. D. Cosmo and S. Zacchiroli, “The Software Heritage Open Science Ecosystem,” in *Software Ecosystems*, Springer International Publishing, 2023, pp. 33–61. doi: 10.1007/978-3-031-36060-2_2.

[122] Software Heritage Archive, “Software Heritage Archive Website.” [Online]. Available: https://archive.softwareheritage.org/

[123] GitHub, “Generated SBOM files will now include a package URL when a manifest file includes a range.” [Online]. Available: https://github.blog/changelog/2024-06-11-generated-sbom-files-will-now-include-a-package-url-when-a-manifest-file-includes-a-range/

[124] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, “A Survey of Flaky Tests,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, Oct. 2021, doi: 10.1145/3476105.

[125] “Nixpkgs committer.” [Online]. Available: https://github.com/orgs/NixOS/teams/nixpkgs-committers?query=drupol